Algoritmo para a Redução de Identificadores em Sistemas OLAP

Relatório de Estágio

Marco António Ferreira Domingues
(2018013362)

Orientadores:

Prof. Jorge Bernardino | ISEC

Prof. Rodrigo Rocha Silva | CISUC

Licenciatura em Engenharia Informática

Ramo de Desenvolvimento de Aplicações

Instituto Politécnico de Coimbra

Instituto Superior de Engenharia de Coimbra

Setembro de 2021

AGRADECIMENTOS

Gostaria de agradecer aos professores Jorge Bernardino e Rodrigo Rocha Silva pela paciência, apoio e disponibilidade durante todo o estágio.

Agradeço também à CISUC pela confiança dada aquando me aceitaram para realizar este estágio.

Gostaria também de agradecer ao ISEC, os seus professores e outros colaboradores por todo o conhecimento, disponibilidade, experiência e momentos partilhados, ainda que muitos deles à distância.

Não poderia deixar de agradecer à minha família, especialmente ao meu pai, Carlos Domingues, e mãe, Maria do Céu Domingues, pela oportunidade que me deram de poder prosseguir estudos, à minha irmã, Rita Domingues, por conseguir abrilhantar dias mais negros e a todo o resto da família mais afastada por me moldarem naquilo em que me tornei.

Por final, gostaria de agradecer aos meus amigos mais próximos, em especial, Gonçalo, Miguel, Nuno e João, por serem mais do que aquilo que posso expressar.

RESUMO

Com o aumento exponencial da quantidade de informação que é produzida e recolhida em todo mundo, a computação dessa informação em sistemas OLAP está a tornar-se cada vez mais dispendiosa relativamente à quantidade de recursos necessário para guardar e processar a mesma. Este problema emergente é cada vez mais sério e está a fazer com que a comunidade científica esteja, cada vez mais, a alocar recursos para encontrar estruturas de dados e algoritmos que permitam solucionar o mesmo total ou parcialmente.

Este estágio, realizado remotamente no CISUC (Centro de Informática e Sistemas da Universidade de Coimbra), sediada em Coimbra, tem por objetivo a criação de uma estrutura e algoritmo que, recorrendo à redução de IDs de registos no algoritmo de Frag-Cubing, permita reduzir a quantidade de memória necessária para guardar e processar *data set*s com tamanhos elevados, permitindo operações OLAP nos mesmos. Foi experimentalmente demonstrado que o algoritmo criado, denominado de *Compressed Inverted Index Data Cube* (CiiCube), conseguiu reduzir eficientemente a quantidade de memória necessária para indexar e processar cubos de dados. Notavelmente, CiiCube conseguiu indexar um *data set* com 500 milhões de registos transformando-o num cubo de dados com apenas 11.99 gigabytes. Frag-Cubing, utilizado como comparação, não conseguiu realizar essa operação tendo a seu dispor os 30 gigabytes de memória principal do mesmo sistema.

Neste relatório é apresentada detalhadamente cada uma das fases realizadas, desde a averiguação do estado da arte até aos resultados obtidos pela estrutura e algoritmo criados, assim como a indicação sobre todas as tecnologias e técnicas usadas.

Palavras-chave: Big Data, Frag-Cubing, *Data Set*, Tabelas de Índices Invertidos, Compressão.

ABSTRACT

With the exponential increase in the amount of information that is produced and collected worldwide, computing such amounts of information in OLAP systems is becoming increasingly expensive relatively to the amount of resources needed to store and process it. This emerging problem is increasingly serious and is causing the scientific community to increasingly allocate resources to find data structures and algorithms that allow to solve it totally or partially.

This internship, carried out remotely at CISUC (Center for Informatics and Systems of the University of Coimbra), based in Coimbra, aims to create a structure and algorithm that, using the reduction of record ids in the Frag-Cubing algorithm, allows to reduce the amount of memory needed to store and process *data sets* with large sizes, allowing OLAP operations on them. It was experimentally demonstrated that the created algorithm, called Compressed Inverted Index Data Cube (CiiCube), managed to efficiently reduce the amount of memory needed to index and process data cubes. Notably, CiiCube was able to index a data set with 500 million records into a data cube with just 11.99 gigabytes. Frag-Cubing, used as a comparison, was not able to perform this operation using the 30 gigabytes of main memory available on the same system.

This report presents in detail each of the phases carried out, from the investigation of the state of the art to the results obtained by the structure and algorithm created, as well as an indication of all the technologies and techniques used.

Keywords: Big Data, Frag-Cubing, *Data Set*, Inverted Index Table, Compression.

ÍNDICE

Agradecimentos i

Resumo iii

Abstract v

Índice vii

Índice de figuras xi

Siglas e acrónimos xvii

1 Introdução 1

1.1 ISEC 1

1.2 CISUC 1

1.3 Objetivos de estágio 2

1.4 Estrutura do relatório 2

2 Gestão e plano de trabalho 3

2.1 Reuniões de orientação 3

2.1 Plano de trabalhos 3

3 Conceitos 5

3.1 OLAP 5

3.2 Bases de dados do tipo *data warehouse* 5

3.3 Algoritmo de índices invertidos 6

3.4 Cubos de dados 7

3.5 Algoritmo Frag-Cubing 8

4 Trabalhos relacionados 13

5 Algoritmos desenvolvidos 17

5.1 Implementação do algoritmo Frag-Cubing 17

5.2 Proposta do algoritmo CiiCube 20

5.2.1 Explicação da compressão 20
5.2.2 Estrutura 1: utilização de matrizes com 3 dimensões 21
5.2.3 Estrutura 2: utilização da classe *DIntArray* 22
5.2.4 Estrutura 3: otimização da classe *DIntArray* 24
5.2.5 Algoritmo de indexação e compressão de registos 25
5.2.6 Algoritmo de interseção para classe *DIntArray* 26
5.2.7 Algoritmo para a realização de *point queries* 29
5.2.8 Algoritmo para a realização de *subcube queries* 29
5.2.9 Estrutura de atualização 30
6 Experiências com CiiCube 35
6.1 Metodologia e ambiente de teste 35
6.2 Características dos *data sets* 36
6.3 Tipos de testes realizados 37
6.4 Avaliação experimental dos *data sets* 38
6.4.1 *Data sets* artificiais 38
6.4.1.1 Testes de indexação 39
6.4.1.2 Testes de *point queries* 44
6.4.1.3 Testes de *subcube queries* 45
6.4.2 *Data sets* reais 51
6.4.2.1 Data sets utilizados 51
6.4.2.2 *Data set Forest Covertype* 51
6.4.2.3 *Data set Connect-4* 54
6.4.2.4 Data set Exame 55
6.5 Conclusões sobre o CiiCube tendo em conta os resultados 56
6.6 Escrita do artigo científico 57
7 Conclusões e trabalho futuro 59
Referências 61

Anexos 63
Anexo A: Proposta de Estágio A-1
Anexo B: Artigo científico 5
Abstract 5
1. Introduction 5
2. Related Work 7
3. The CiiCube Approach 9
3.1. CiiCube Representation 10
3.2. CiiCube's Data Cube Indexation 11
3.3. Intersection Algorithm B-13
3.4. Update Algorithm B-17
3.5. Queries Implemented B-19
4. Experiments B-20
4.1. Experimental Setup B-20
4.2. Indexation Runtimes and Structure Sizes B-21
4.3. Point Queries B-23
4.4. Subcube queries B-24
4.5. Tests With Real World Data B-26
4.5.1. Data Sets Used B-26
4.5.2. Data Set Forest Covertype B-26
4.5.3. Data Set Exame B-29
4.5.4. Data Set Connect-4 ..... B-30
5. Conclusions and future work B-31

ÍNDICE DE FIGURAS

Figura 1. Representação Visual Simplificada do Trabalhado do Processo ETL Durante a Fase de Adição de Dados Numa Data Warehouse....6
Figura 2. Exemplo Representativo de um Cubo de Dados [9] 7
Figura 3. Exemplo de uma Lista de registos compostos por um TID e 3 Atributos....9
Figura 4. Relação Entre Cubo de Dados, Shell-Fragment e Listas Invertidas em Frag-Cubing.....9
Figura 5. Conjunto de Bitmaps por Atributo de valor baseado na Figura 3....13
Figura 6. Exemplo do Algoritmo de bCubing Utilizando o Exemplo da Figura 3....15
Figura 7. Modelo Entidade Relacionamento do Algoritmo de Frag-Cubing 18
Figura 8. Algoritmo de Interseção Final de Frag-Cubing....20
Figura 9. Lista de valores para uma dimensão....21
Figura 10. Exemplo de estrutura usando matriz tridimensional 22
Figura 11. Exemplo de estrutura usando a classe inicial DIntArray 23
Figura 12. Exemplo de estrutura usando a segunda iteração da classe DIntArray....24
Figura 13. Algoritmo de Indexação e Compressão de CiiCube 25
Figura 14. Algoritmo de Interseção de CiiCube....28
Figura 15. Algoritmo para realizar point queries....29
Figura 16. Algoritmo para realizar subcube queries....30
Figura 17. Algoritmo de Atualização de Registos....32
Figura 18. Algoritmo de Recuperação de Listas de TIDs Através de um Valor de Atributo....33
Figura 19. Tempo de Indexação, em Segundos, Variando a Cardinalidade do Data Set com T = 10M, D = 30, S = 1.5 e C = 2500, 5000, 7500 e 10000 39
Figura 20. Tempo de Indexação, em Segundos, Variando o Número de Registos do Data Set com D = 30, C = 2500, S = 1.5 e T = 10M, 30M, 50M, 70M e 90M....40
Figura 21. Tempo de Indexação, em Segundos, Variando o Número de Dimensões do Data Set com T = 10M, C = 2500, S = 1.5 e D = 30, 50 e 63....40
Figura 22. Tempo de Indexação, em Segundos, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500 e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5....41
Figura 23. Tamanho da Estrutura, em Gigabytes, Variando a Cardinalidade do data set com T = 10M, D = 30, S = 1.5 e C = 2500, 5000, 7500 e 10000 42
Figura 24. Tamanho da Estrutura, em Gigabytes, Variando o Número de Registos do Data Set com D = 30, C = 2500, S = 1.5 e T = 10M, 30M, 60M e 90M....42

Figura 25. Tamanho da Estrutura, em Gigabytes, Variando o Número de Dimensões do Data Set com T = 10M, C = 2500, S = 1.5 e D = 30, 50 e 63.....43
Figura 26. Tamanho da Estrutura, em Gigabytes, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500 e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5.....43
Figura 27. Tempo de Processamento de Point Queries Variando a Skew em Data Sets com T = 130M, D = 30, C = 2500, 30 Operadores Equal e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5.....45
Figura 28. Tempo de Processamento DE Subcube Queries, em Segundos, Variando a Cardinalidade do Data Set com T = 10M, D = 30, S = 1.5, 1 Operadores Equal e 2 Operadores Inquire e C = 2500, 5000, 7500 e 10000. 46
Figura 29. Tempo de Processamento em Subcube Queries, em Segundos, Variando o Número de Registos do Data Set com D = 30, C = 2500, 1 Operadores Equal, 2 Operadores Inquire e S = 1.5 e T = 10M, 30M, 50M, 70M e 90M. 47
Figura 30. Tempo de Processamento em Subcube Queries, em Segundos, Variando o Número de Dimensões do Data Set com T = 10M, C = 2500, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e D = 30, 50 e 63. 47
Figura 31. Tempo de Processamento em Subcube Queries, em Segundos, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500, 1 Operadores Equal, 2 Operadores Inquire e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5. 48
Figura 32. Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com T = 10M, D = 30, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e C = 2500, 5000, 7500 e 10000 49
Figura 33. Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com D = 30, C = 2500, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e T = 10M, 30M, 50M, 70M e 90M.....49
Figura 34. Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com T = 10M, C = 2500, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e D = 30, 50 e 63 50
Figura 35. Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500, 1 Operadores Equal, 2 Operadores Inquire e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 e 5.....50
Figura 36. Tempo de Indexação e Tamanho da Estrutura para o Data Set Forest Covtype 52
Figura 37. Tempo de processamento e memória utilizada para uma subcube query com três operadores inquire sobre as dimensões com cardinalidade 2, 2 e 7, pertencentes ao data set Forest Covertype 52

Figura 38. Tempo de processamento e memória utilizada para uma subcube query com três operadores inquire sobre as dimensões com cardinalidade 3858, 1398 e 801, pertencentes ao data set Forest Covertype 53
Figura 39. Tempo de processamento e memória utilizada para uma subcube query com quatro operadores inquire sobre as dimensões com cardinalidade 3858, 801, 2 e 2, pertencentes ao data set Forest Covertype 53
Figura 40. Tempo de Indexação e Tamanho da Estrutura final para o Data Set Connect-4 54
Figura 41. Tempo de processamento e utilização de memória para subcube queries com 1 operador equal e 10 operadores inquire sobre o data set Connect-4.....55
Figura 42. Tempo de Indexação e Tamanho da Estrutura Final para o Data Set Exame .55
Figura 43. Tempo de processamento e utilização de memória para subcube queries com 1 operador equal e 2 operadores inquire sobre o data set Exame 56

SIGLAS E ACRÓNIMOS

CBI	Compressed Bitmap Index
CISUC	Centro de Informática e Sistemas da Universidade de Coimbra
DEIS	Departamento de Engenharia Informática e de Sistemas
ETL	Extract, Transform, Load
ISEC	Instituto Superior de Engenharia de Coimbra
LEI	Licenciatura em Engenharia Informática
OLAP	Online Analytical Processing
SQL	Structured Query Language
TID	Tuple Identifier/ Id de Registo

1 INTRODUÇÃO

Este relatório descreve o trabalho realizado no âmbito do estágio integrado na Licenciatura em Engenharia Informática (LEI) realizada no Departamento de Informática e Sistemas (DEIS), pertencente ao Instituto Superior de Engenharia de Coimbra (ISEC) no segundo semestre do ano letivo de 2020/2021.

O estágio foi realizado entre março e setembro de 2021 para o CISUC, tendo sido realizado à distância devido à pandemia por Covid-19.

O CISUC é um centro de investigação nacional que se encontra intimamente conectado com a investigação e desenvolvimento. No decorrer de investigação realizada anteriormente foi realizada a proposta para um algoritmo que utilizasse compressão para reduzir a quantidade de memória necessária para guardar cubos de dados.

Nesse seguimento, o CISUC propôs este estágio, que consiste no desenvolvimento da estrutura de compressão e escrita de um artigo científico sobre a mesma.

1.1 ISEC

O Instituto Superior de Engenharia de Coimbra (ISEC), é uma unidade orgânica pertencente ao Instituto Politécnico de Coimbra (IPC), que em 1974 resultou da conversão do antigo Instituto Industrial e Comercial de Coimbra. Em 1988 foi incorporado no Instituto Politécnico de Coimbra (IPC) [1].

O ISEC tem como missão "*a criação, transmissão e difusão de cultura, ciência e tecnologia*" com o objetivo de formar profissionais capazes e de promover o desenvolvimento da região em que se insere [2].

Atualmente, o ISEC oferece 39 cursos, que incluem licenciaturas, mestrados e cursos técnicos superiores profissionais, nos quais se inserem um total de 3357 alunos, 174 docentes e investigares, assim como 83 profissionais não docentes [3].

1.2 CISUC

O Centro de Informática e Sistemas da Universidade de Coimbra (CISUC) é um centro de investigação nas áreas de informática e comunicações. Criada em 1991, o centro trabalha num vasto leque de tópicos relacionados com a ciência da computação e engenharia.

Os objetivos da empresa passam por fazer pesquisa e desenvolvimento inovadores, treinar jovens altamente qualificados para a área de pesquisa e

desenvolvimento, cooperar em projetos nacionais e internacionais, assim, como promover a disseminação de resultados através de contratos com diferentes empresas [4].

1.3 Objetivos de estágio

Foram indicados dois objetivos principais para o estágio. O primeiro objetivo foi a criação de uma estrutura para redução do consumo de memória. O segundo objetivo indicado foi a realização de um algoritmo de atualização sobre os dados guardados na estrutura criada para o primeiro objetivo.

A junção destes dois objetivos com a realização de testes sobre os mesmos permitiria a escrita de um artigo científico onde a estrutura criada e resultados de testes sobre a mesma seriam publicados.

1.4 Estrutura do relatório

Este relatório está subdividido da seguinte forma:

No capítulo 2 é realizada a apresentação do plano de trabalhos seguido no estágio. No capítulo 3 são indicados conceitos essenciais para o entendimento do tema abordado durante o estágio. No capítulo 4 é apresentada a pesquisa realizada sobre outros algoritmos similares àquele que foi desenvolvido. No capítulo 5 são indicadas as diferentes arquiteturas testadas e o raciocínio realizado até se chegar à estrutura e algoritmos finais. No capítulo 6 é apresentada a metodologia de testes, resultados dos mesmos e as conclusões obtidas. No capítulo 7 são apresentadas as conclusões e possíveis trabalhos futuros.

2 GESTÃO E PLANO DE TRABALHO

Tanto o CISUC como o ISEC realizaram múltiplas iniciativas de apoio aos estagiários com o objetivo de orientar o trabalho a realizar.

2.1 Reuniões de orientação

Durante o estágio foram realizadas várias reuniões de orientação. Inicialmente, as reuniões de estágio foram realizadas a cada duas semanas, sendo que posteriormente passaram a ser realizadas semanalmente.

Os participantes das reuniões eram o orientador do ISEC, orientador do CISUC e outro colega que realizava um estágio no mesmo centro.

Durante as reuniões, os estagiários apresentavam o trabalho realizado desde a última reunião realizada, sendo que ambos os orientadores davam indicações quanto à qualidade do trabalho realizado e quanto ao trabalho a realizar até à reunião seguinte.

Estas reuniões serviram também para que os estagiários pudessem colocar dúvidas que não poderiam ser solucionadas por outros meios de contacto, devido à natureza das mesmas.

2.1 Plano de trabalhos

A proposta inicial de estágio, que se encontra na sua integra Anexo A, divide o plano de trabalhos em 6 tópicos:

1. Pesquisa bibliográfica;
2. Testes com abordagens clássicas;
3. Desenvolvimento da abordagem;
4. Escrita de um *position paper*;
5. Análise de resultados;
6. Elaboração do relatório e do artigo.

Os 5 primeiros tópicos são realizados sequencialmente, sendo que a sexta parte é realizada ao longo de todo o projeto.

Durante a realização do estágio, o tópico 4 não foi realizado, uma vez que os orientadores não acharam necessária a escrita do *position paper* sobre a abordagem realizada. Todos os outros tópicos de trabalho foram efetuados e serão descritos em baixo.

A primeira tarefa realizada foi a pesquisa bibliográfica. Esta tarefa resumiu-se na pesquisa de bases de conhecimento para o trabalho a realizar. Esta tarefa foi realizada durante as duas primeiras semanas do estágio. Mais pormenores sobre a mesma poderão ser vistos no capítulo 3 e 4, sendo os mesmos o contexto tecnológico e trabalhos correlatos.

A segunda tarefa foi a realização de testes com abordagens clássicas. Esta tarefa resumiu-se na escrita de uma implementação do algoritmo Frag-Cubing, assim como a realização de alguns testes sobre o mesmo. Esta tarefa teve uma duração de 4 semanas. Mais pormenores sobre a implementação realizada poderão ser vistos no subcapítulo 5.1. Deve notar-se que, mesmo após esta tarefa estar completa, foram realizados melhoramentos à implementação realizada.

A terceira tarefa realizada foi o desenvolvimento e implementação de uma abordagem onde fosse realizada a redução de ids e consequente algoritmo de atualização. Esta tarefa teve uma duração de 8 semanas. Pormenores sobre as abordagens criadas e a evolução das mesmas, assim como uma explicação pormenorizada dos algoritmos mais significativos da abordagem final podem ser vistos ao longo do subcapítulo 5.2.

A quarta tarefa foi a realização de testes sobre a abordagem realizada e a análise dos resultados obtidos. Esta teve duração de 8 semanas, sendo que teve início antes do fim da terceira tarefa. Pormenores sobre a metodologia de testes, testes realizados e conclusões sobre os mesmos podem ser vistos no capítulo 6.

No final de todos os testes serem realizados, foi investido tempo na escrita do artigo científico, cujo processo de poderá ser visto no subcapítulo 6.6. O artigo resultante encontra-se em anexo.

Esta lista de tarefas permitiu à criação do algoritmo de redução de ids, nomeado de *"Compressed Inverted Index Data Cube*", ou CiiCube, um algoritmo que, como será visto, consegue reduzir consideravelmente a quantidade de memória utilizada para o processamento de cubos de dados.

3 CONCEITOS

Antes de se poder fazer qualquer estudo num determinado tema é necessária a obtenção de boas bases na área. Tendo isso em conta, a primeira tarefa realizada passou por pesquisa que permitisse compreender todo o contexto tecnológico à volta do algoritmo a ser desenvolvido.

3.1 OLAP

OLAP (Online Analytical Processing) é a capacidade de "consolidar, ver e analisar informação de acordo com múltiplas dimensões, de forma a que faça sentido para um ou mais analistas empresariais específicos" [5].

Esta tecnologia é especialmente utilizada nas áreas empresariais, uma vez que permite que um grande conjunto de dados possa ser visto de uma forma simples. Devido à sua utilidade, esta tecnologia é também utilizada em estudos dos mais variados temas, uma vez que permite mostrar a mesma informação de pontos de vista diferentes.

A capacidade de realizar operações OLAP é uma característica essencial nos programas utilizados para a tomada de decisões baseadas na informação disponível.

3.2 Bases de dados do tipo *data warehouse*

A capacidade de OLAP acima descrita é, geralmente, realizada sobre grandes repositórios de dados, como é o caso de *Data warehouses*.

Data warehouse são repositórios centrais de dados com origem em uma ou múltiplas fontes. Este tipo de sistemas é utilizado tanto para fazer um registo histórico sobre um conjunto de dados como também, para permitir análises sobre a informação contida.

O processo mais comum para introdução de dados numa *data warehouse* é denominado *ETL (Extract, Transform, Load)*. Como os dados de uma *data warehouse* são obtidos de fontes diferentes, podem ter características distintas. O algoritmo *ETL* é utilizado para fazer uma uniformização dos dados antes de armazenar os mesmos na *data warehouse* facilitando as análises realizadas sobre os mesmos. A *Figura 1* ilustra o processo de introdução de dados numa *data warehouse* utilizando o algoritmo *ETL*.

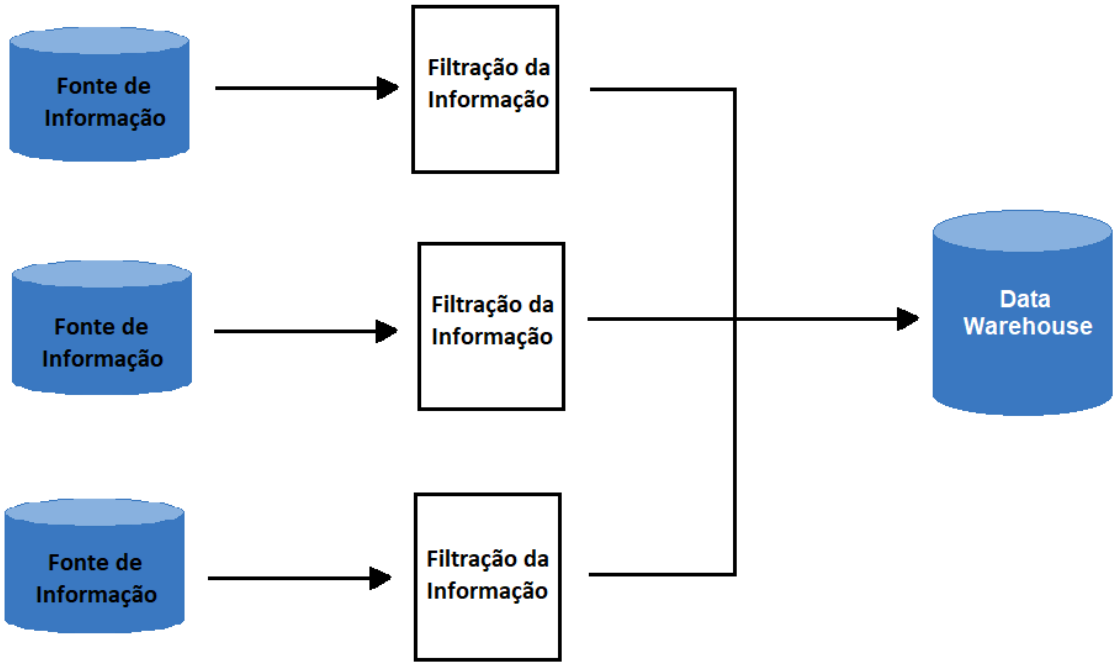


Figura 1. *Representação Visual Simplificada do Trabalhado do Processo ETL Durante a Fase de Adição de Dados Numa Data Warehouse*

Muitas vezes *data warehouses* são carregadas em algoritmos com capacidade de realizarem operações OLAP, permitindo assim que os dados contidos nas mesmas sejam analisados.

3.3 Algoritmo de índices invertidos

Outro conceito pertinente para a contexto deste estágio é a ideia de índices invertidos, desenvolvida para mapear o conteúdo existente baseado na sua localização [6]. Numa base de dados normal, a informação é contida numa lista de registos, sendo cada registo caracterizado por um identificador único e um conjunto de atributos. Utilizando o algoritmo de índices invertidos, os registos mantêm um identificador e atributos, porém, ao invés de se guardar uma lista de registos, o algoritmo guarda os dados de forma a que cada atributo diferente tenha uma lista de identificadores de registos associada.

Esta maneira distinta de representar os dados permite a que, aquando se quer obter uma lista de registos que contêm determinado atributo, ao invés de se ter de verificar registo a registo se contém o atributo procurado pode-se, simplesmente, devolver a lista de identificadores de registo associada ao atributo em questão.

Este tipo de estrutura e algumas propriedades que a mesma acaba por possuir serão fundamentais para alguns algoritmos que permitam operações OLAP, como é o caso de Frag-Cubing, que será infra explicado.

3.4 Cubos de dados

Cubos de dados são operadores criados com o objetivo de superar as limitações do operador *group-by* quanto à capacidade de relacionar dados multidimensionais de forma eficiente [7]. Cubos de dados são representados através de *arrays* multidimensionais que, geralmente, guardam referências para informação cujo tamanho é muito superior àquele que pode ser representado na memória principal do computador [8].

A informação guardada em cubos de dados é guardada ao longo das múltiplas dimensões de uma forma arbitrária sendo, por vezes, possível aumentar o nível de precisão com que os dados podem ser representados nos mesmos.

Cubos de dados são utilizados para representar informação ao longo de múltiplas medidas de interesse, permitindo pesquisas completas e rápidas sobre os dados guardados, utilizando diferentes perspetivas, sendo, dessa forma, uma estrutura extremamente útil onde se podem realizar análises de dados.

Na *Figura 2* é possível ver um exemplo de cubo de dados.

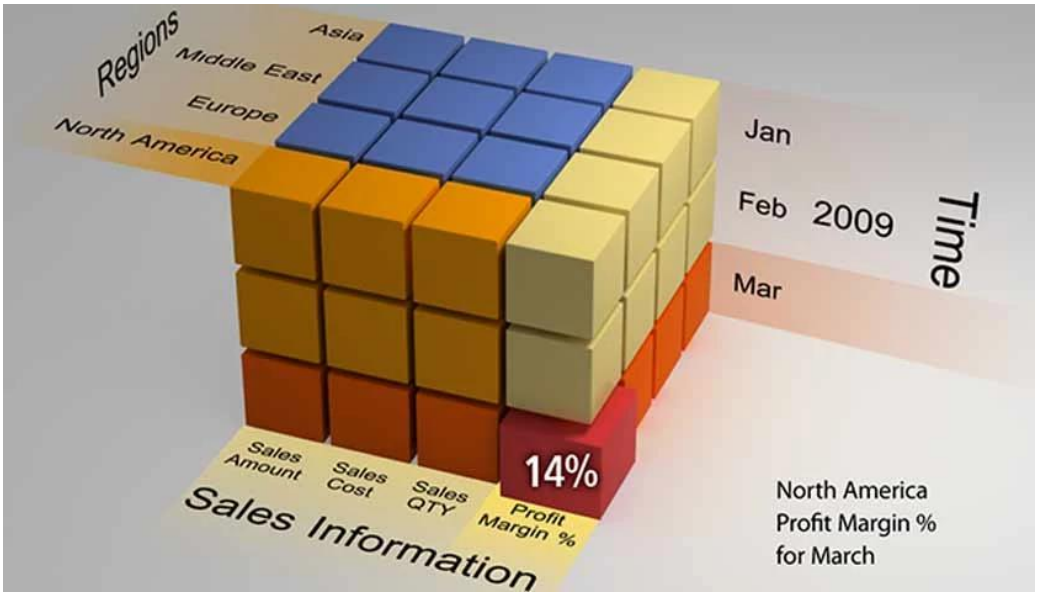


Figura 2. *Exemplo Representativo de um Cubo de Dados [9]*

Olhando para o exemplo da *Figura 2*, é possível ver 3 dimensões distintas: regiões, informação de vendas e tempo. Cada uma das dimensões está representada ao longo de cada face, sendo que as faces estão divididas conforme os valores de atributo

das respetivas dimensões. Para obter os dados de qualquer valor de atributo basta apenas aceder à devida parte do cubo.

O processamento de um cubo de dados é um problema com crescimento exponencial tanto a nível do tempo de processamento quanto à utilização de memória. Formalmente, assumindo um cubo de dados com D dimensões que têm C cardinalidade, o processamento completo desse cubo de dados teria de processar D^{C+1} combinações.

3.5 Algoritmo Frag-Cubing

Realizar a computação de cubos de dados é o equivalente a fazer um pré-processamento de todas as análises possíveis sobre o mesmo, pelo que essa pré-computação traz a vantagem de que as operações sobre os cubos de dados realizadas sejam extremamente rápidas. Porém, a quantidade de memória necessária para guardar um cubo de dados completo tem um crescimento exponencial consoante o número de dimensões existentes, pelo que não é uma solução viável para cubos de dados com uma tamanho moderado ou elevado [10].

Frag-Cubing [10] é um algoritmo que permite a computação de operações sobre conjuntos de registos de dados, denominados de *data sets*. Este algoritmo apresentou múltiplas novidades relativas à estrutura e forma como os dados eram guardados em memória, nomeadamente, a utilização de guardar os dados em índices invertidos e a subdivisão do cubo de dados em *Shell-Fragments*. Uma vez que o processamento completo de um cubo de dados é, devido ao seu crescimento exponencial, impossível de ser realizado, Frag-Cubing propôs a ideia de transformar o cubo de dados em múltiplos cuboides, mais pequenos, que fazem o processamento completo das dimensões que contêm, reduzindo a quantidade de tempo e memória para guardar os cuboides e, ainda assim, garantindo um bom desempenho no processamento de pedidos.

Os registos, também denominados de *tuples*, são, no algoritmo de Frag-Cubing, guardados em índices invertidos. A hierarquia existente neste algoritmo é bastante simples: os registos são caracterizados por um identificador, TID, e um conjunto de atributos, como pode ser visto na *Figura 3*. Frag-Cubing cria um conjunto de listas e associa cada uma das mesmas a um dos atributos. Após isso, é feita uma leitura dos registos, sendo o TID relativo ao registo adicionado à lista de TID associada aos atributos que o mesmo tenha. Note-se que os atributos estão divididos por múltiplas dimensões, sendo cada dimensão uma característica pela qual um atributo é definido.

TID	DIMENSÃO 1	DIMENSÃO 2	DIMENSÃO 3
1	A1	B1	C1
2	A2	B1	C2
3	A1	B2	C3
4	A3	B3	C1
5	A1	B1	C1

Figura 3. *Exemplo de uma Lista de registos compostos por um TID e 3 Atributos*

Como fazer computação completa de cubos de dados não é uma opção viável relativamente ao tamanho da estrutura resultante e tempo de processamento necessários. Para solucionar esse problema, Frag-Cubing propõem a ideia de *Shell-Fragments*. Um *Shell-Fragment* é um subcubo de dados, isto é, assumindo um cubo de dados com um determinado número de dimensões, um *Shell-Fragment* é uma estrutura responsável por processar uma quantidade parcial desse número de dimensões, podendo ser visto como um cubo de dados parcial, ou cuboide. Para realizar operações OLAP sobre atributos de diferentes dimensões e diferentes cuboides, Frag-Cubing utiliza um método que intercepta as listas de TIDs obtidas de diferentes cuboides e, dessa forma, obtém os registos que possuem os atributos pedidos. Note-se que, no ambiente da pesquisa a realizar neste estágio, serão utilizados *Shell-Fragments* que guardam uma única dimensão.

De um ponto de vista estrutural, Frag-Cubing é composto por três partes distintas:

I. As listas de TIDs que guardam os TIDs invertidos para um determinado atributo;
II. Os *Shell-Fragments*, que, no contexto da nossa pesquisa representam uma única dimensão e que contêm os atributos e respetivas listas de TIDs;
III. O cubo de dados, que é composto pelos vários *Shell-Fragments*.

Na *Figura 4* pode ser vista uma ilustração da estrutura de Frag-Cubing indicada acima.

Figura 4. *Relação Entre Cubo de Dados, Shell-Fragment e Listas Invertidas em Frag-Cubing.*

O algoritmo Frag-Cubing consegue obter respostas a *queries* realizadas sobre o subcubo. A estrutura da *query* é muito simples: o utilizador indica, para cada uma das dimensões do subcubo, um operador que indica a operação a realizar sobre a dimensão. Os operadores podem ser de três tipos distintos [10]:

- Operador igual (***equal)***: este operador indica ao algoritmo que, para responder à *query* a ser realizada, procura-se ter em conta apenas os registos que, na dimensão em que este operador se encontra, tenham o valor a ser instanciado pelo operador. Este operador é representado pelo valor que o utilizador colocar;
- Operador **agregado**: este operador indica ao algoritmo que, para aquela dimensão, não é necessário ter em conta qualquer valor, pelo que realizar ações sobre a dimensão pode ser ignorado. Este operador é representado pelo símbolo '*';
- Operador inquirido (***inquire)***: este operador indica imediatamente que a *query* a ser realizada é do tipo subcubo, que será explicado adiante. Este operador é representado pelo valor '?'.

As *queries* a ser realizadas pelo algoritmo podem ter dois tipos distintos, *point queries* e *subcube queries*, que serão pormenorizadamente explicadas de seguida:

Point Queries são *queries* que visam responder à contagem (em SQL: *count*) do número de registos que contêm os atributos instanciados. Estas *queries* são compostas apenas por operadores *equal* e agregado.

Os procedimentos a realizar passam pela obtenção das listas de TIDs relativas aos atributos instanciados através dos operadores *equal* e realizar a interseção das mesmas.

Para realizar a interseção de duas listas de TIDs invertidas existe uma função de interseção. Esse método é responsável em devolver uma lista composta pelos TIDs que se encontram em comum em duas listas de TIDs.

A função de interseção é um elemento fundamental para o desempenho do algoritmo, acabando por ser necessária a adoção das técnicas mais rápidas que forem conseguidas. Note-se que, se nenhuma dimensão for instanciada, o algoritmo devolve o número de registos que se encontram no cubo de dados, não realizando nenhuma operação de interseção.

São operações relativamente simples e rápidas, pelo que são muito usadas, no contexto deste estágio, para determinar a velocidade da função de interseção do algoritmo, consecutivamente, permitindo retirar conclusões sobre o desempenho do mesmo.

Subcube queries são *queries* que visam fazer análises a uma parte do cubo de dados, também chamado de subcubo. A análise de um determinado subcubo é feita

através de *point queries* a todas as possibilidades existentes de diferentes *queries* tendo em conta as dimensões instanciadas e inquiridas.

Quando o algoritmo deteta a existência de um operador *inquire* na *query* colocada pelo utilizador, imediatamente começa a realizar um subcubo. O subcubo é composto pelas dimensões que foram inquiridas e contem, para essas dimensões, listas de TIDs com os registos cujos atributos são aqueles que foram instanciados na *queries* pela utilização de operadores *equal*. Após a criação do subcubo, o algoritmo realiza todas as *point queries* possíveis, tendo em conta as diferentes combinações de atributos para as dimensões inquiridas [10].

Esta operação é considerada complexa e lenta, sendo que foi verificado durante o estágio realizado que, em alguns casos mais extremos, a adição da operação de mostrar o resultado no ecrã pode adicionar horas à mesma. Esta operação pode ser utilizada, principalmente, para avaliar a comportamento do algoritmo quanto à utilização de memória na resposta a *queries*. Para além disso, pequenas perdas de desempenho existentes sobre o algoritmo de interseção podem tornar-se extremamente óbvias, uma vez que a operação de interseção pode ser realizada milhões de vezes durante uma subcube query.

4 TRABALHOS RELACIONADOS

Ainda que a base para o trabalho a ser realizado seja o algoritmo de Frag-Cubing, a pesquisa por outros algoritmos da mesma área permite a que existe um maior envolvimento com a mesma, podendo até criar condições onde é possível utilizar o conhecimento adquirido em outros algoritmos para melhorar aquele que esta a ser desenvolvido.

Tendo isso em conta, a compreensão de outros trabalhos da área é uma excelente fonte de conhecimento de outras técnicas utilizadas para o mesmo problema.

Existem duas abordagens principais para algoritmos de computação de cubos de dados: a utilização de *bitmaps* e utilização de listas invertidas.

Relativamente à utilização de *bitmaps* para representação de registos, dois trabalhos interessantes são *BitCube* [11] e *Compressed Bitmap Index* [12].

BitCube [12] é um algoritmo que, para cada valor de atributo, cria um *bitmap* que permite mapear quais os registos que correspondem ao valor a ser listado. O *bitmap* é composto por valores binários, *bits*, cuja posição do bit nos bitmaps são utilizadas para identificar o registo. Isto faz com que todos os *bitmaps* tenham o mesmo tamanho, sendo a mesma posição nos vários *bitmaps* referentes a um único registo. Por sua vez, os *bits* podem ter dois valores: 1 ou 0. Quando o registo contém o atributo de valor associado ao *bitmap* em que se encontra, então recebe o valor 1, caso contrário recebe o valor 0. Para obter a lista de registos que contém um determinado número de atributos, o algoritmo de *BitCube* começa por obter os bitmaps correspondentes aos atributos e realiza uma simples interseção dos mesmos.

Na *Figura 5* é possível ver o conjunto de *bitmaps* para cada valor. Esta figura é baseada na lista de registos que se encontra na *Figura 3.*

ATRIBUTO	BITMAP
A1	10101
A2	01010
B1	11001
B2	00100
B3	00010
C1	10011
C2	01100

Figura 5. *Conjunto de Bitmaps por Atributo de valor baseado na Figura 3*

Compressed Bitmap Index (CBI) [12]é um algoritmo que funciona de uma forma muito semelhante a *BitCube*, sendo que simplesmente acrescenta compressão aos *bitmaps*. Em vez dos *bitmaps* guardarem todos os *bits*, o algoritmo de CBI guarda a primeira e última posições em que um *bit* com o valor 1 aparece no *bitmap*. Para além disso, o *bitmap* passa a representar apenas os registos que estejam dentro das primeira e última posições com *bit* cujo valor é 1.

Relativamente à utilização de índices invertidos para representar cubos de dados, para além de Frag-Cubing, que já foi introduzido no capítulo 3, existem outros trabalhos interessantes, nomeadamente, qCube [13], bCubing [14] e H-FRAG [15].

qCube [13] é uma extensão do algoritmo de Frag-Cubing. Este algoritmo adiciona a capacidade de resposta a *range queries*, implementando um novo operador denominado *range operator*. Tal como Frag-Cubing, este algoritmo utiliza uniões e interseções para processar as *queries*, sendo que também tem um consumo de memória e tempo de processamentos com crescimento linear conforme o crescimento do número de registos e de atributos por registo.

H-FRAG [15] é um algoritmo que utiliza memória híbrida para guardar os registos, distribuindo as listas invertidas entre o disco e a memória principal do sistema. Quando a operação de indexação está a ocorrer, o algoritmo H-FRAG começa por decidir quais as listas invertidas que são guardadas em memória e quais são guardadas em disco. Para isso, o H-FRAG começa por analisar todo o *data set* de forma a obter a frequência de cada um dos valores de atributo. De seguida, o algoritmo calcula a frequência média dos atributos. As listas invertidas cuja frequência dos seus atributos seja menor que a frequência média são guardadas no disco, sendo que aquelas cuja frequência é maior do que a frequência média são guardadas na memória principal. O processo de realização de *queries* sobre o cubo de dados é igual ao apresentado no Frag-Cubing, com o pormenor extra de que, em alguns casos, é necessária a obtenção de listas invertidas do disco. Este trabalho conseguiu muito sucesso na redução do consumo de memória, porém, existe um aumento no tempo de processamento necessário para obter listas invertidas que se encontrem no disco, tempo de processamento que não existe quando as listas invertidas se encontram previamente em memória.

bCubing [9] é também um algoritmo que utiliza tanto disco como memória principal, tal como H-FRAG, porém, a gestão da memória híbrida é completamente diferente. A maior diferença entre bCubing e outros trabalhos que utilizam índices invertidos, tal como Frag-Cubing, é a estrutura. Enquanto que a maior parte dos algoritmos utilizam uma única tabela onde têm acesso a todas as listas invertidas, bCubing utiliza duas tabelas diferentes: uma para guardar as listas invertidas e outra para mapear a distribuição dos atributos ao longo da primeira tabela, permitindo um acesso mais eficiente à mesma. Os registos encontram-se divididos em blocos. Todos os blocos têm o

mesmo tamanho à exceção do último que pode ter menos registos por não haver suficientes para o completar. A primeira tabela é guardada em disco. Nela, são construídas, para cada bloco, listas invertidas para os valores de atributo que aparecem nos registos guardados no bloco. A segunda tabela, mantida em memória, é usada para mapear a frequência com que os valores de atributo aparecem em cada bloco, dessa forma, o algoritmo sabe se um bloco contém registos com determinado atributo de valor sem a necessidade de aceder ao mesmo. Testes realizados com este algoritmo mostraram que o mesmo torna-se muito mais eficiente do que Frag-Cubing para processar cubos de dados com tamanhos maciços, sendo, por exemplo, capaz de anexar um *data set* com 10^9 registos num sistema com 128 gigabytes de memória, sendo que não foi possível a Frag-Cubing realizar tal operação no mesmo sistema. Deve também sublinhar-se que o algoritmo tem também um tempo de processamento superior a Frag-Cubing, o que é esperado uma vez que utiliza memória híbrida.

Na *Figura 6* pode ser visto o exemplo da *Figura 3* usado em bCubing, com blocos com tamanho 3.

Tablela 1:

DIMENSÃO	BID	ATRIBUTO	LISTA DE TIDS
1	1	A1	1,3
		A2	2
	2	A1	4
		A3	5
2	1	B1	1,2
		B2	3
	2	B1	5
		B3	4
3	1	C1	1
		C2	2
		C3	3
	2	C1	4,5

Tabela 2:

DIMENSÃO	ATRIBUTO	BID	NÚMERO DE TIDS
1	A1	1	2
		2	1
	A2	1	1
	A3	2	1
2	B1	1	2
		2	1
	B2	1	1
	B3	2	1
3	C1	1	1
		2	1,2
	C2	1	1
	C3	1	1

Figura 6. Exemplo do Algoritmo de bCubing Utilizando o Exemplo da Figura 3

A Figura 6. é composta por duas tabelas diferentes, *Tabela 1* e *Tabela 2*. A *tabela 1* guarda as listas invertidas de TIDs por bloco, sendo guardada em disco. A *tabela 2* mapeia a frequência os atributos, por dimensão, ao longo dos vários blocos.

5 ALGORITMOS DESENVOLVIDOS

Após a obtenção de conhecimento sólido sobre a área, assim como outras técnicas utilizadas com o objetivo de resolver problemas similares, o passo seguinte foi o desenvolvimento e implementação dos algoritmos.

Como o algoritmo desenvolvido teve por base o algoritmo de Frag-Cubing, a primeira implementação realizada foi do algoritmo de Frag-Cubing. Após essa implementação inicial, começaram a desenvolver-se as várias ideias e implementações do algoritmo de compressão.

5.1 Implementação do algoritmo Frag-Cubing

A primeira decisão a ser realizada em qualquer projeto deste tipo é a linguagem a ser utilizada. O CISUC deu liberdade para poder escolher a linguagem a utilizar na escrita do código. A linguagem escolhida foi Java pelo facto de ser simples e efetiva para o trabalho a realizar.

Como já foi indicado anteriormente, a estrutura do algoritmo de Frag-Cubing pode ser dividida em três partes distintas: o cubo de dados, os *shell-fragments* e as listas para guardar os TIDs invertidos. Tendo em conta essa estrutura foram definidas duas classes para gerir o cubo: *ShellFragment* e *DataCube*.

A classe *DataCube* é o ponto de entrada do cubo de dados. Esta é simplesmente composta por um *array* de objetos da classe *ShellFragment*, denominado *ShellFragmentList*. Deve notar-se que, ao contrário de um algoritmo completo de Frag-Cubing, não foi adicionada a capacidade de utilizar um único objeto *ShellFragment* para guardar múltiplas dimensões, pelo não é necessária nenhuma estrutura extra para mapear as dimensões ao longo dos objetos *ShellFragment*¸ já que cada um dos mesmos pode ser considerado uma dimensão.

A classe *ShellFragment* é composta por uma matriz de inteiros, denominada "*matrix"*, que guarda aos TIDs invertidos para cada valor. Essa matriz representa um valor diferente em cada linha e guarda os TIDs dos registos correspondentes nas colunas da mesma. Este processo está relacionado com a estrutura dos *data sets* e será explicado posteriormente. Para além disso, os objetos desta classe têm também um array de inteiros, denominado "*size*", utilizado para determinar a quantidade de TIDs para cada atributo guardado no objeto "*matrix*", e dois inteiros que guardam o valor mais baixo e mais alto a ser guardado, respetivamente "*lower*" e "*upper*".

Estas duas classes permitem criar o algoritmo Frag-Cubing de uma forma bastante simples. A *Figura 7* mostra a forma como as classes *DataCube* e *ShellFragment* se relacionam.

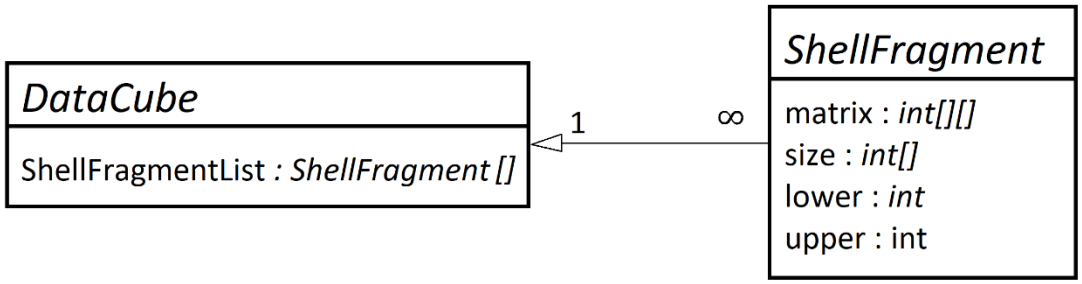


Figura 7. *Modelo Entidade Relacionamento do Algoritmo de Frag-Cubing*

Com o objetivo de ter uma referência para o resultado final do algoritmo, foi disponibilizado um executável de Frag-Cubing. O algoritmo recebe como input um *data set* e um determinado tipo de input para realizar as *queries*, sendo que após as realizar mostrava o output das mesmas. A velocidade do algoritmo, seja a indexar o *data set*, transformando-o num cubo de dados, como a responder a *queries*, servia também como base para o desempenho esperado.

A estrutura do *data set* era bastante simples: Na primeira linha encontravam-se, em primeiro lugar, o número de registos que o *data set* contém seguido do maior valor existente no *data set*, para cada uma das dimensões dos registos. Seguidamente, cada registo era colocado numa linha distinta.

O facto dos *data sets* indicarem o maior valor existente para cada dimensão, acrescentado ao facto do valor mínimo de cada dimensão ser *1*, permite a que se crie uma versão muito simplista de uma *hash table* para representar os valores. Uma *hash table* é uma estrutura que, geralmente, contém uma chave associada a um objeto, sendo a sua maior vantagem o facto de haver uma complexidade constante relativamente ao tempo de acesso ao objeto através da chave. No caso da matriz "*matrix*" existente nos objetos da classe *ShellFragment*, a chave é o valor de atributo a ser guardado e o objeto a ser guardado é o array que guarda os TIDs para o seu respetivo valor.

A versão simplista de *hash table* realizada utilizava o facto de se saber o valor mínimo e máximo a serem guardados, colocando na matriz "*matrix*" o número de linhas necessário para representar todos os valores. Os TIDs relativos a cada valor eram colocados nas colunas das matrizes. Deve notar-se que a linguagem Java utiliza, nativamente, como matrizes, uma estrutura chamada *jagged arrays*. A estrutura *jagged arrays* difere de matrizes normais no sentido em que todos os *arrays* são unidimensionais, sendo que, cada linha de uma matriz é, na verdade, um ponteiro para outro *array*. Esta

estrutura tem a vantagem de permitir que matrizes tenham colunas com tamanhos diferentes, porém tem a desvantagem de utilizar uma quantidade exponencial de memória, como será demonstrado posteriormente.

Deve notar-se também que a linguagem Java contém também, nativamente, uma estrutura to tipo *hash table*, porém, essa estrutura contém capacidades que não são necessárias para o algoritmo a ser escrito e que, por isso, utilizam uma quantidade de memória superior e velocidade inferior, pelo que se não foi utilizada. Outro ponto similar é a utilização de tipos nativos, com array de inteiros, em contraste com classes de memória dinâmica, como é o caso da classe *List*, a utilização deste tipo de classes com elevado nível de abstração tem a vantagem de facilitar a escrita do código, porém utiliza uma quantidade elevada de memória quando comparado com a utilização de tipos nativos, pelo que, durante o trabalho, foram sempre utilizados os tipos básicos existentes em Java.

A escrita deste algoritmo teve várias fases de evolução diferentes, isto é, ainda que o algoritmo realizasse o seu trabalho, não significava estar bom, uma vez que, no início era relativamente lento. O aumentar da velocidade do algoritmo de Frag-Cubing realizado em Java foi algo progressivo e que necessitou de bastante pesquisa e imaginação para reduzir os tempos de processamento.

Um bom exemplo que reflete as melhorias realizadas foi o método de interseção. O método de interseção, com já foi indicado acima, é uma parte fundamental do algoritmo, sendo o principal ponto onde a velocidade de Frag-Cubing pode ser limitada, uma vez que este é utilizado múltiplas vezes e tem uma natureza extensa.

O primeiro método de interseção realizado utilizava dois *loops for* para obter e comparar os TIDs pertencentes a dois atributos. Tendo em conta que os TIDs encontram-se em ordem numéria nas listas de TIDs invertidos, este método podia ser consideravelmente melhorado. Para o melhorar, utilizava-se um único *loop while*, que tinha uma complexidade linear, ao contrário da utilização de dois *loops for*, que tinham complexidade temporal exponencial. Na *Figura 8* é possível verificar o pseudocódigo do algoritmo de interseção no seu estado final.

```
Input: array arrayA and array arrayB
Output: array arrayC;

1.      cA = 0;
2.      cB = 0;
3.      While cA < arrayA.length and  cB < arrayB.length do
4.              if arrayA[cA] equal arrayB[cB] then
5.                      Adds the value arrayA[cA] to arrayC and increments cA and cB;
6.              else if arrayA[cA] < arrayB[cB] then
7.                      increments cA;
8.              else
9.                      increments cB;
10.             end
11.     end
```

Figura 8. *Algoritmo de Interseção Final de Frag-Cubing*

Este simples mudança a nível da função de interseção fez com que, o algoritmo de Frag-Cubing feito em Java tivesse tempos de resposta a *point queries* similares ao algoritmo de Frag-Cubing que foi disponibilizado pelo CISUC.

5.2 Proposta do algoritmo CiiCube

O nome dado ao algoritmo de compressão criado foi "*Compressed Inverted Index Data Cube*", ou CiiCube. O processo realizado até à obtenção desse algoritmo foi complexo, longo e constantemente evolutivo, pelo que neste subcapítulo é apresentada toda a história e explicação técnica essenciais para a compreensão do mesmo.

5.2.1 Explicação da compressão

A estrutura e algoritmos a implementar tentam diminuir o consumo da memória necessária para guardar um cubo de dados através da compressão de *ranges* de TIDs, isto é, quando existe um conjunto de TIDs seguidos, não é necessário guardar todos para os representar, ao invés disso, pode representar-se esse intervalo através do menor e maior valores do mesmo.

Para facilitar a visualização, imagine-se a lista de TIDs sem compressão *n = {1, 2, 3, 4, 5, 6, 7, 18, 27, 28, 29, 30, 33}*. No algoritmo de Frag-Cubing, este conjunto de TIDs é guardado num array tal como se encontra em cima: *[1, 2, 3, 4, 5, 6, 7, 18, 27, 28, 29, 30, 33]*. Com o algoritmo de compressão que se pretende utilizar, os intervalos de valores podem ser substituídos pelos valores máximos e mínimos dos mesmos, sendo que esta lista de TIDs fica da seguinte forma: *[[1, 7], [18], [27, 30], [33]]*. Esta forma de compressão fez com que se passasse a representar a lista com apenas 5 valores, quando antes a mesma era composta por 13.

Este tipo de compressão é, em última instância, altamente dependente como os registos se encontram distribuídos ao longo do *data set*, isto porque é teoricamente possível que o algoritmo não consiga realizar qualquer compressão em *data sets* com um tamanho aceitável, porém isso é uma possibilidade teórica que, como será demonstrado futuramente, não tem qualquer semelhança com a realidade.

Deve notar-se ainda que os intervalos são realizados quando existem três ou mais TIDs seguidos na lista, o que faz com que não seja realizada compressão quando apenas existem dois valores seguidos. Como exemplo, a lista de TIDs *[1, 2]* não sofre qualquer compressão, enquanto que a lista *[1, 2, 3]* já passa a ser comprimida para *[1, 3]*.

5.2.2 Estrutura 1: utilização de matrizes com 3 dimensões

A ideia mais básica para implementação deste tipo de compressão passou pela implementação da modificação da matriz "*matrix*", com duas dimensões, que se encontra na classe *ShellFragment*, transformando-a numa matriz com três dimensões. A primeira dimensão, isto é, as linhas da matriz, continuariam a corresponder a cada um dos possíveis atributos de valor para a dimensão em causa. A segunda dimensão, informalmente nominada de coluna, e que antes guardava apenas um único TID, agora passa a guardar uma de duas coisas: ou um TID, ou um intervalo de TIDs.

A distinção entre valor único de intervalo de valores é realizada através do tamanho do array no qual a "coluna" em questão aponta, isto é, se tiver tamanho de 1, então guarda um valor, se, por outro lado, tiver o tamanho de 2, guarda um intervalo de valores.

TID	ATRIBUTO
1	1
2	1
3	1
4	1
5	2
6	1
7	2
8	2
9	2
10	2
11	1
12	1
13	2
14	1
15	1
16	1

Figura 9. *Lista de valores para uma dimensão*

Para visualizar mais facilmente aquilo que se esta a explicar, vejamos o seguinte exemplo:

Na *Figura 9* é possível visualizar uma lista de TIDs e o seu respetivo atributo. Deve notar-se que esta não é a estrutura de um verdadeiro *data set*, sendo apenas um exemplo simplificado.

Esta lista é composta por 16 registos, sendo que podem ter dois atributos diferentes, *a* ou *b*, sendo, portanto, a cardinalidade de 2. Como a cardinalidade é de dois, então a estrutura "*matrix*" da classe *ShellFragment* tem duas linhas.

Baseado no exemplo acima, é possível retirar duas listas de TIDs, uma para cada atributo de valor. Para o atributo 1 tem-se a lista de TIDs: 1 = *[1, 2, 3, 4, 6, 11, 12, 14, 15, 16]* e para o atributo 2 tem-se: 2 = *[5, 7, 8, 9, 10, 13]*. Na *Figura 10* é possível ver a forma como a utilização de matrizes tridimensionais funcionaria.

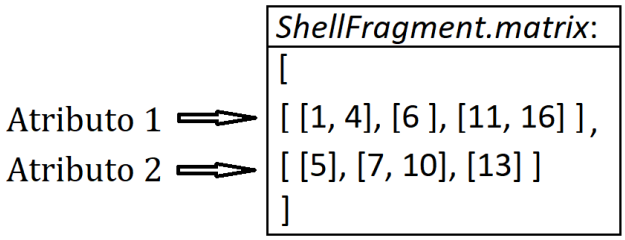


Figura 10. *Exemplo de estrutura usando matriz tridimensional*

A *Figura 10* demonstra a estrutura de *arrays* resultante para guardar o *data set* de exemplo indicado na *Figura 9*. Do lado direito da figura estão, em cada linha, a lista de TIDs para cada atributo, estando em cada coluna um array que guarda, ou um TID, ou um intervalo de TIDs representado pelo seu maior e menor valores. Do lado esquerdo da imagem, encontram-se "legendas" que indicam o atributo a que cada linha pertence.

Como já foi acima referido, a linguagem Java utiliza nativamente *jagged arrays* para definir matrizes. Este tipo de estrutura faz com que arrays multidimensionais possam ser vistos como "*arrays* de *arrays*". Este tipo de estrutura faz com que exista uma maior flexibilidade para o tamanho dos arrays, porém tem um maior peso de memória. Tal como classes, o objeto array tem um peso base de 24 bytes, utilizando mais 4 bytes por cada inteiro que guardar. Ora, vale notar que na estrutura acima, cada coluna tem um array que guarda, ou um inteiro, ou dois inteiros, pelo que, nestes arrays, utiliza-se mais memória para os objetos arrays do que para guardar os TIDs.

Notando também que cada carater '[' na imagem é representativo de um novo array, é compreensível que esta solução não é eficaz. Alguns testes informais realizados através da indexação de alguns *data sets* demonstraram que esta estrutura utiliza mais memória do que a estrutura de Frag-Cubing.

5.2.3 Estrutura 2: utilização da classe *DIntArray*

A primeira estrutura apresentada guardava as listas de TIDs comprimidas segundo a ideia de redução proposta, porém a mesma falhou em reduzir a quantidade de memória utilizada pelo facto de que a quantidade de arrays existentes ser extremamente elevada. Tendo em conta isso, a solução para conseguir reduzir a memória é apenas uma: reduzir a quantidade de arrays existente.

Olhando de novo para a estrutura, é possível verificar que os TIDs são guardados de uma de duas formas:

- O TID encontra-se sozinho, representado por um inteiro;
- O TID encontra-se num intervalo, representado por dois inteiros;

Com isto em conta decidiu criar-se uma nova classe: *DIntArray*. Esta classe *DIntArray* é composta por dois arrays distintos, *array1* e *array2*. O array de inteiros array*1* é responsável por guardar ou os TIDs que não fazem parte de intervalos, ou o menor valor de um intervalo. Por outro lado, o array de inteiros *array2* guarda, ou o valor mais elevado de um intervalo, ou então o valor -1, quando não existe nenhum intervalo. Os valores entre os dois arrays encontra-se interligados através da sua posição no array, também denominada de *index*.

A mudança de utilização de simples arrays para uma classe obrigou também a mudanças a nível da classe *ShellFragment*. Agora, o objeto "*matrix*", ao invés de ser um array de inteiros, passa a ser um array de objetos da classe *DIntArray*.

Deve notar-se também que, agora, a classe *ShellFragment* já não tem o array "*sizes*", uma vez que cada classe *DIntArray* tem o seu próprio mecanismo para calcular a utilização dos arrays internos, que será aqui ignorada para efeitos de simplificação.

Voltando a utilizar o *data set* de exemplo da *Figura 9*, pode ver-se de seguida a ilustração desta segunda estrutura na *Figura 11*.

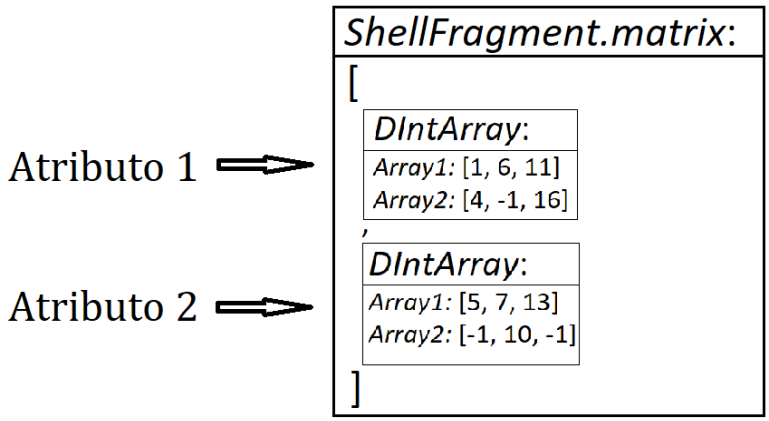


Figura 11. *Exemplo de estrutura usando a classe inicial DIntArray*

A estrutura acima delineada resolveu o problema da criação de quantidades maciças de arrays e, em alguns testes informais realizados, conseguiu até reduzir a quantidade de memória utilizada.

Apesar deste claro avanço, a estrutura indicada tem um problema simples: não é suficientemente flexível para permitir poupar memória. A falta de flexibilidade é bastante visível quando se pensa no pior caso. O pior caso é um *data set* para o qual não é possível fazer qualquer compressão. Ora, um *data set* desse tipo resultaria nesta estrutura utilizar

exatamente o dobro da memória que Frag-Cubing utilizaria, isto porque cada o objeto *array2* que todas as classes *DIntArray* contêm seria inutilizado.

5.2.4 Estrutura 3: otimização da classe *DIntArray*

Tendo em conta a iteração anterior da classe *DIntArray*, o ponto seguinte de evolução realizado foi a tentativa de evitar o espaço perdido com os valores não comprimidos.

A nova ideia para melhorar a compressão do algoritmo passou pela diferenciação entre os intervalos de TIDs comprimidos e os TIDs que não puderam ser comprimidos. Esta mudança requer três arrays diferentes: dois para guardar os TIDs comprimidos da mesma forma que já se fazia e um novo para se guardar TIDs que não puderam ser comprimidos.

A classe *DIntArray* é agora composta pelos arrays *reducedPos1*, *reducedPos2* e *noReductionArray*. O array *reducedPos1* é responsável por guardar os menores valores de intervalos de TIDs. O array *reducedPos2* é, por outro lado, responsável por guardar os maiores valores de intervalos de TIDs. Os TIDs que não podem ser comprimidos são guardados no array *noReductionArray*.

Utilizando ainda o exemplo da *Figura 9*, pode ver-se na *Figura 12* o resultado obtido com esta nova estrutura:

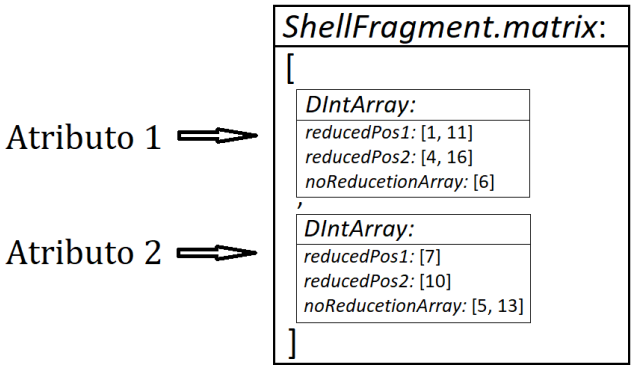


Figura 12. *Exemplo de estrutura usando a segunda iteração da classe DIntArray*

Com esta nova iteração da classe *DIntArray* não existe qualquer desperdício de memória. Testes informais realizados com a mesma permitiram concluir que a compressão era suficientemente sólida para que esta estrutura possa ser a final.

A partir deste ponto, qualquer referência à classe *DIntArray* terá esta estrutura de compressão mixa em mente.

5.2.5 Algoritmo de indexação e compressão de registos

A compressão de TIDs é realizada durante a indexação do *data set*, isto é, à medida que o cubo de dados é construído.

Existem três operações diferentes durante a adição de um novo TID ao cubo de dados: o novo TID pode criar um novo intervalo, o novo TID pode fazer parte de um intervalo já existente, ou, o novo TID não sofre qualquer compressão.

Quando um novo TID é adicionado começa-se por verificar se o mesmo pertence a um intervalo já existente. Para realizar essa verificação basta verificar que o novo TID é o valor seguinte do maior TID pertencente ao último intervalo criado. Caso seja o caso, é apenas necessário substituir o maior TID do último intervalo criado, que se encontra no array *reducedPos2*, pelo novo TID.

Caso o novo TID não pertença a um intervalo já criado, como visto em cima, verifica-se se é possível criar um novo intervalo utilizando o novo TID. Para realizar isso duas condições necessitam de ser satisfeitas: a primeira é o array *noReductionArray* ter dois ou mais TIDs inseridos e a segunda passa pelos dois últimos TIDs inseridos nesse mesmo array, juntamente com o novo TID, serem valores numericamente seguidos. Caso isto aconteça, então removem-se os dois últimos TIDs do array *noReductionArray* e cria-se um novo intervalo, adicionando o penúltimo valor que se encontrava no array sem compressão e o novo TID a novas posições dos arrays, respetivamente, *reducedPos1* e *reducedPos2*.

Se nenhum dos casos acima descritos for possível, então não é possível fazer qualquer compressão, pelo que o novo TID é simplesmente adicionado à última posição do array *noReductionArray*.

Na *Figura 13* é possível ver o pseudocódigo para o algoritmo de indexação e compressão de registos.

```
Input: newTid to be stored; sizeReduced which is a variable that points to the next unused position of the arrays reducedPos1 and reducedPos2;
sizeNonReduced which is a variable that points to the next unused position of the array noReductionArray; reducedPos1 which stores the firt element of an
interval; reducedPos2 which stores the last element of an interval; noReductionArray which stores non compressed tids;
Output: none

1.  if sizeReduced > 0 and  reducedPos2[sizeReduced - 1] + 1 equal newTid then
2.      reducedPos2[sizeReduced - 1] = newTid;
3.  else if sizeNonReduced > 2 and noReductionArray[sizeNonReduced - 1] + 1 equalthen  id and noReductionArray[sizeNonReduced - 2] + 2 equal newTid then
4.      reducedPos1[sizeReduced] = noReductionArray[sizeNonReduced - 2];
5.      reducedPos2[sizeReduced] = newTid;
6.      sizeReduced = sizeReduced + 1;
7.      sizeNonReduced = sizeNonReduced - 2;
8.  else
9.      noReductionArray[sizeNonReduced] = newTid;
10.     sizeNonReduced = sizeNonReduced + 1;
11. end
```

Figura 13. *Algoritmo de Indexação e Compressão de CiiCube*

Por motivos de simplificar a explicação dos algoritmos não é tomada em conta a necessidade de aumentar o tamanho dos arrays *noReductionArray, reducedPos1* e *reducedPos2*. Porém, é necessário acrescentar que, quando estes arrays não têm mais espaço para adicionar novos TIDs e existe a necessidade de adicionar um novo TID que ocupa um novo espaço, um novo array com o dobro do tamanho do anterior é criado e os TIDs são copiados para o novo array, que passa a ter o papel do *array* completo anterior. Deve sublinhar-se que esta operação de aumento do tamanho dos arrays é, por sua natureza, uma operação lenta, sendo que diminuir a quantidade de vezes que a mesma é realizada pode trazer melhorias significativas nos tempos de indexação.

5.2.6 Algoritmo de interseção para classe *DIntArray*

Deve sublinhar-se que a mudança da estrutura na representação de dados refletiu-se também em mudanças nos métodos que interagem com os mesmos, nomeadamente, o método de interseção, que é utilizado para responder a *queries*, como já foi indicado anteriormente.

O método de interseção de dois arrays utilizado no algoritmo de Frag-Cubing, que foi explicado anteriormente, aproveita o facto dos TIDs se encontrarem em ordem numérica ao longos dos arrays para intercetar os mesmos com um único *loop*. Este método de interseção é o mais eficiente que se conhece, pelo que o método de interseção para a classe DIntArray deve seguir um algoritmo similar.

O método de interseção pode ser algoritmicamente dividido em duas fases distintas: a escolha do TID ou intervalo de TIDs que será intercetado e a interseção entre os TIDs escolhidos na fase anterior.

A primeira fase, de escolha de TIDs, realizada individualmente em cada classe DIntArray a ser comparada, passa pela escolha do menor TID válido para comparação nos arrays *reducedPos1* e *noReductionArray*. Para definir qual o menor TID válido utilizam-se variáveis externas que serão usadas para indicar a posição do menor valor válido de cada um dos dois arrays. O menor TID válido de ambos os arrays é aquele que será usado na segunda fase para "representar" a classe em que se encontra. Vale notar que, caso o TID escolhido pertença ao array *reducedPos1*, a fase seguinte utiliza o intervalo que definido pelo TID escolhido desse array e o TID que se encontra na mesma posição do array *reducedPos2*, como foi indicado acima.

A segunda fase passa pela comparação dos valores escolhidos em cada uma das classes *DIntArray* e definição do próximos menores TIDs válidos para a iteração seguinte. Note-se que a comparação pode ser feita entre dois TIDs, dois intervalos de TIDs ou um TID e um intervalo de TID.

Quando a comparação é realizada entre dois TIDs, simplesmente verifica-se se os mesmos são iguais. Se ambos forem iguais adiciona-se o TID ao objeto *DIntArray* responsável por guardar o resultado da interseção e aumenta-se o ponteiro secundário para que o TID seguinte dos arrays seja o menor valor válido. Caso sejam diferentes, então aumenta-se o ponteiro do menor TID dos dois representantes.

Quando a comparação é realizada entre um TID e um intervalo de TIDs, verifica-se se o TID se encontra dentro do intervalo, sendo que em caso afirmativo adiciona-se o mesmo ao objeto *DIntArray* que guarda o resultado da operação de interseção. Caso o TID não esteja dentro do intervalo de TIDs, verifica-se, entre o TID e o menor valor do intervalo, qual o valor menor, aumentando o ponteiro secundário que aponta para o menor valor máximo do mesmo.

Quando a comparação é realizada entre dois intervalos de TIDs verifica-se se o menor valor de um dos intervalos encontra-se dentro do outro intervalo de TIDs. Se esse for o caso, verifica-se se o menor valor é igual ao maior valor do outro intervalo, sendo que, se sim, então adiciona-se esse valor, caso contrário, então adiciona-se, ao objeto DIntArray resultante, um intervalo composto pelo TID que se encontra entre o outro intervalo e o menor TID que seja máximo dos dois intervalos. De seguida, o intervalo de TIDs com o menor "maior TID" tem o seu ponteiro secundário aumentado.

O processo de interseção composto pelas duas fases acima descritas continua até que um dos objetos *DIntArray* a serem comparados não tenha mais nenhum TID ou intervalo de TIDs disponíveis para comparar.

Na *Figura 14* é possível ver pseudocódigo para o algoritmo de interseção de CiiCube, descrito acima.

```
Input: DIntArray DA and DIntArray DB;
Output: DIntArray DC;
Input: DIntArray DA and DIntArray DB;
Output: DIntArray DC;

1.  aNonreduced=0;
2.  aReduced=0;
3.  bNonreduced=0;
4.  bReduced=0;
5.  While DA or DB have TIDs or TID intervals to intersect do
6.    if DA.sizeNonReduced equal aNonreduced or DA.reducedPos1[aReduced] < DA.sizeNonReduced[aNonReduced] then
7.      if DB.noReductionArray equal bNonreduced or DB.reducedPos1[bReduced] < DB.noReductionArray[bNonReduced] then
8.        if DA.reducedPos1[aReduced] >= DB.reducedPos1[bReduced] and DA.reducedPos1[aReduced] <= DB.reducedPos2[bReduced] then
9.          if DA.reducedPos1[aReduced] equal DB.reducedPos2[bReduced] then
10.           adds the value DA.reducedPos1[aReduced] to DC and increments aReducted and bReducted;
11.         else
12.           if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
13.             adds the interval [DA.reducedPos1[aReduced]; DA.reducedPos2[aReduced]] to DC and increments aReduced;
14.           else
15.             adds the interval [DA.reducedPos1[aReduced]; DB.reducedPos2[bReduced]] to DC and increments bReduced;
16.           end
17.         end
18.       else if DB.reducedPos1[aReduced] >= DA.reducedPos1[bReduced] and DB.reducedPos1[aReduced] <= DA.reducedPos2[bReduced] then
19.         if DB.reducedPos1[bReduced] equal DA.reducedPos2[aReduced] then
20.           adds the value DB.reducedPos1[bReduced] to DC and increments aReducted and bReducted;
21.         else
22.           if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
23.             adds the interval [DB.reducedPos1[bReduced]; DA.reducedPos2[aReduced]] to DC and increments aReduced;
24.           else
25.             adds the interval [DB.reducedPos1[bReduced]; DB.reducedPos2[bReduced]] to DC and increments bReduced;
26.           end
27.         end
28        else if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
29            increments aReduced;
30        else
31            increments bReduced;
32.       end
33.     else if DB.reducedPos1 equal bReduced or DB.reducedPos1[bReduced] > DB.noReductionArray[bNonReduced] then
34.       if DB.noReductionArray[bNonReduced] >= DA.reducedPos1[aReduced] and DB.noReductionArray[bNonReduced] <= DA.reducedPos1[aReduced] then
35.         adds the value DB.noReductionArray[bNonReduced] to DC and increments the counter bNonReduced;
36.       else if DB.noReductionArray[bNonReduced] <= DA.reducedPos2[aReduced] then
37.         increments bNonReduced;
38.       else
39.         increments aReduced;
40.       end
41.     end
42.   else if DA.reducedPos1 equal aReduced or DA.reducedPos1[aReduced] > DA.noReductionArray[aNonReduced] then
43.     if DB.noReductionArray equal bNonreduced or DB.reducedPos1[bReduced] < DB.noReductionArray[bNonReduced] then
44.       if DA.noReductionArray[aReduced] >= DB.reducedPos1[bReduced] and DA.noReductionArray[aReduced] <= DB.reducedPos1[bReduced] then
45.         adds the value DA.noReductionArray[aNonReduced] to DC and increments aNonReduced;
46.       else if DA.noReductionArray[aNonReduced] <= DB.reducedPos2[bReduced] then
47.         increments aNonReduced;
48.       else
49.         increments bReduced;
50.       end
51.     else if DB.reducedPos1 equal bReduced or DB.reducedPos1[bReduced] > DB.noReductionArray[bNonReduced] then
52.       if DB.noReductionArray[bNonReduced] equal DA.noReductionArray[aNonReduced] then
53.         adds the value DA.noReductionArray[aNonReduced] to DC and increments bNonReduced and aNonReduced;
54.       else DB.noReductionArray[bNonReduced] < DA.noReductionArray[aNonReduced] then
55.         increments bNonReduced;
56.       else
57.         increments aNonReduced;
58.       end
59.     end
60.   end
61. end
```

Figura 14. *Algoritmo de Interseção de CiiCube*

5.2.7 Algoritmo para a realização de *point queries*

Uma *point query* é o tipo de pesquisa mais simples que o algoritmo realizado permite fazer. Tal como já foi dito anteriormente, uma *point query* visa responder o *count* do número de registos que contêm as características definidas na *query*.

Para obter o *count* de registos que contêm os atributos definidos através de operadores *equal*, o algoritmo começa a obter e guardar num array as classes *DIntArray* que contêm as listas invertidas para cada um dos atributos instanciados com esses operadores.

Após isso, são então realizadas as operações de interseção, realizada entre pares de listas invertidas e utilizando o algoritmo de interseção acima explicado, das quais surge uma instância da classe *DIntArray* com os TIDs resultantes.

Finalmente, é devolvida a classe DIntArray composta pelos TIDs resultantes. Deve notar-se que o programa mostra no ecrã o número de TIDs que se encontra na classe DIntArray devolvida.

Na *Figura 15* encontra-se o pseudocódigo do algoritmo para realização de *point queries*. Deve notar-se que, à exceção da utilização da classe *DIntArray*, o pseudocódigo mostrado nesta figura é igual para CiiCube e Frag-Cubing.

```
Input: query being made
Output: DIntArray A

1.  DIntArray[] B;
2.  for each equal operator in query then
3.      obtains the TID list related to the instantiated attribute and adds it to DIntArray B;
4.  DIntArray A = DIntArray B [0];
5.  for each DIntArray object in DIntArray B[] then
6.      DIntArray A = intersect DIntArray A with DIntArray object;
```

Figura 15. *Algoritmo para realizar point queries*

5.2.8 Algoritmo para a realização de *subcube queries*

Subcube queries são utilizadas para fazer análises detalhadas de secções do subcubo, pelo que são consideradas operações complexas e demoradas. O algoritmo utilizado para a realização das mesmas tem como objetivo reduzir o tempo de processamento para cubos de dados com tamanhos muito elevados.

O algoritmo começa por obter a lista de TIDs que serão utilizados na *subcube query*, sendo que para isso, troca, momentaneamente, todos os operadores *inquire* por operadores agregado, já que *point queries* não aceitam operadores *inquire*. Para além disso, o operador *inquire* não limita os atributos de valor pedidos, sendo nesse sentido,

equivalentes a operadores agregado. De seguida, obtém a lista de TIDs resultante através do mesmo método utilizado para realizar *point queries*.

Após isso, o algoritmo cria um subcubo. Este subcubo é composto pelos TIDs obtidos no passo anterior, sendo que, neste subcubo, são apenas representadas as dimensões inquiridas na *query* realizada.

O terceiro e último passo passa pela realização de todas as possibilidades de *point queries* a realizar sobre este subcubo, onde existe um *loop* que define as *point queries* a realizar e fá-las sobre o cubo. As *point queries* realizadas, assim como o seu resultado, são guardadas numa matriz, sendo que a totalidade da matriz, isto é, o resultado da *point query* realizada, poderá ou não ser mostrada, conforme a opção "verbose" existente no algoritmo.

Na *Figura 16* é possível ver o pseudocódigo do algoritmo utilizado para a realização de *subcube queries*.

```
Input: the query being made
Output: none

1.    changedQuery = change inquired operators to aggregate operators in query;
2.    DIntArray A = point query sending changedQuery;
3.    Creates subcube composed by DIntArray A and inquired dimensions in query;
4.    int[][] matrix;
5.    while there is a different point query available do
6.          DIntArray B = point query sending the possibility of query;
7.          stores in matrix the possibility of query and the number of TIDs in DIntArray B;
8.    end
9.    if verbose is true then
10.         shows matrix;
```

Figura 16. *Algoritmo para realizar subcube queries*

5.2.9 Estrutura de atualização

Foi também desenhado uma estrutura que permitisse a atualização de dados sem a necessidade do cálculo total ou parcial do cubo de dados. Esta escolha provém do facto de, devido à natureza da utilização deste tipo de algoritmos, não ser esperado que existam quantidades relevantes de atualizações a realizar, pelo que, quando existem operações de atualização, as mesmas devem executadas ser o mais eficientemente possível.

As atualizações a serem realizadas podem ser de dois tipos: adição de um novo registo ou modificação dos atributos de um registo existente. A adição de um novo registo é realizada utilizando mesmo algoritmo que foi explicado no subcapítulo 5.2.5, pelo que a única estrutura necessária é para atualização de atributos de registos já existentes.

A classe *ShellFragment* é responsável por guardar todos os atributos e respetivas listas de TIDs de uma dimensão, pelo que é a classe mais pertinente para se guardar as mudanças atualizações realizadas.

Foram adicionados, à classe *ShellFragment*, dois arrays, *tidModified* e *valueModified*, que, respetivamente, guardam os TIDs de registos cujo atributo foi modificado e o novo valor modificado do registo. Deve esclarecer-se que a adição de registos a esta estrutura é realizada ao nível da dimensão, isto é, um registo que seja modificado a nível de um único atributo terá a modificação realizada e adicionado à estrutura na dimensão à qual o mesmo atributo pertence, pelo que nas dimensões não modificadas não existe qualquer adição do registo a esta nova estrutura.

Uma vez que, com esta estrutura, a obtenção de valores adiciona operações extra de averiguação dos registos modificados e os seus valores, como será explicado pormenorizadamente à frente, os TIDs guardados no array *tidModified* são colocados em ordem numérica.

Quando existe uma modificação de um registo em determinado atributo, o algoritmo começa por averiguar se o novo atributo indicado é igual ao atributo que se encontrava originalmente no cubo de dados quando ocorreu a indexação do mesmo. Se esse for o caso, o algoritmo averigua se o registo em causa foi anteriormente modificado, sendo que em caso positivo remove o TID e anterior atributo, respetivamente, dos arrays *tidModified* e *valueModified*, terminando a operação.

Se o novo atributo for diferente do atributo que se encontrava inicialmente no cubo de dados aquando da realização da indexação, o algoritmo verifica se o registo em causa teve qualquer modificação realizada anteriormente para a dimensão em causa. Em caso afirmativo, então o algoritmo apenas substitui o atributo modificado anterior pelo novo.

Caso o registo não tenha sido modificado anteriormente, o algoritmo coloca o TID e novo atributo nos arrays *tidModified* e *valueModified*, respetivamente, sendo que são colocados em posições que permitam a que se mantenha a regra de que os TIDs guardados no array *tidModified* encontram-se numericamente ordenados.

O pseudocódigo do algoritmo de atualização de registos poderá ser visto na *Figura 17*.

```
Input: the TID being modified and its newValue; tidModified, which is an array that stores the modified tids; valueModified, which
is an array that stores the new values of the modified tids; sizeModified, wich is a numerical variable used to store the number of
modified tids stored.
Output: none;

1.  DataCubeValue = the attribute value of TID in the main data cube;
2.  if newValue equal DataCubeValue then
3.      for each T in tidModified do
4.          if T equal TID then
5.              removes T and the correspondent value in valueModified;
6.              return;
7.          end
8.      end
9.  end
10. for I from 0 to sizeModified do
11.     if tidModified[I] equal TID then
12.         valueModified[I] = newValue;
13.         return;
14.     end
15. end
16. for I from 0 to sizeModified do
17.     if tidModified[I] > TID then
18.         pushes the values with index >= I a single position to the right in the arrays tidModified and valueModified;
19.         tidModified[I] = TID;
20.         valueModified[I] = newValue;
21.         sizeModified = sizeModified + 1;
22.     end
23. end
```

Figura 17. *Algoritmo de Atualização de Registos*

Aquando da existência de registos modificados numa determinada dimensão, o processo de obtenção da lista invertida de TIDs que contêm um determinado atributo tem o passo extra de remover ou adicionar TIDs conforme os registos modificados.

Para fazer tal verificação, o algoritmo tem de averiguar os valores de todos os TIDs que se encontram no array *tidModified* seguido duas regras:

- Se um determinado TID se encontrar na lista de TIDs invertidos associada a um atributo e no array *tidModified*, então esse TID é retirado da lista a devolver, uma vez que o atributo representado do cubo de dados encontra-se desatualizado.
- Se um determinado TID que esteja no array *tidModified* não se encontrar na lista de TIDs invertidos associada a um atributo, então averigua-se se o atributo atualizado é igual ao atributo cujos TIDs são procurados, sendo que, se for o mesmo atributo, o TID é adicionado à lista invertida de TIDs a devolver.

O facto dos TIDs guardados no array *tidModified* encontrarem-se em ordem numérica, tal como os TIDs que se encontram nas listas invertidas associadas a um atributo, permite que se utilize um algoritmo com um único *loop* para realizar todas as operações necessárias.

Na *Figura 18* é possível ver o pseudocódigo do algoritmo de recuperação de listas invertidas de TIDs para um determinado atributo, tendo em conta os registos que foram atualizados.

```
Input: Attribute value at whose list is to be retrieved
Output: DIntArray A containing the TIDs of tuples that contain the attribute value at

1.    DIntArray A = obtains at's TID list from the main data cube
2.    if tidModified.length > 0 then
3.        counter = 0;
4.        while counter < tidModified.length do
5.            if tidModified[counter] is found in DIntArray A then
6.                remove tidModified[counter] from DIntArray A;
7.            else if valueModified[counter] is equal to at then
8.                adds tidModified[counter] to DIntArray A;
9.            end
10.           increment counter;
11.       end
12.   end
```

Figura 18. *Algoritmo de Recuperação de Listas de TIDs Através de um Valor de Atributo*

6 EXPERIÊNCIAS COM CIICUBE

A terceira parte do trabalho a realizar foram os testes formais ao algoritmo desenvolvido. Até ao momento existiram apenas referencias a testes informais. Testes informais são testes realizados em que, ou o *data set* utilizado serve apenas de exemplo para confirmar que todos os resultados se encontram corretos, ou então os resultados obtidos não seguem a metodologia de testes determinado. Em contraste, testes formais são aqueles que seguem uma metodologia bem definida, consistente e que faça sentido para os casos a testar.

Ao longo do estágio foram realizados milhares de testes formais. A metodologia de teste, os resultados sobre os mesmos e um pequeno resumo do processo de escrita do artigo científico poderão ser vistos de seguida.

6.1 Metodologia e ambiente de teste

A metodologia de testes determinada para obter os valores de resultado de todos os testes é a seguinte:

- Realizam-se cinco vezes o mesmo teste, guardando os valores obtidos;
- Remove-se o maior e menor valores das cinco repetições;
- O resultado final é a média aritmética dos três valores restantes.

Esta metodologia é *standard* para a realização de testes sobre o desempenho de programas em Java. Valores estatísticos como o desvio padrão não foram registados uma vez que não são relevantes para este tipo de análise e não são referidos em nenhuma parte por outros trabalhos correlatos.

Para facilitar o cálculo da dos valores de resultado dos testes, um *script* secundário foi desenvolvido para facilitar nesse processo. Para além disso, foram também desenvolvidos outros pequenos *scripts* que permitem a transformação de, por exemplo, bytes para gigabytes, o que é bastante útil quando é necessária a apresentação de dados.

Deve também acrescentar-se que o algoritmo foi reiniciado após cada uma das repetições realizadas, isto porque o algoritmo de *caching* do Java, mecânica nativa da linguagem e que não pode ser desativada, consegue diminuir consideravelmente o tempo de respostas a *queries* quando as mesmas já foram feitas anteriormente, reduzindo o tempo de resposta, em alguns casos, em mais de 50%.

Os testes foram realizados num servidor remoto, com um sistema *Linux*, sendo a ligação ao mesma realizada por *ssh*, utilizando um algoritmo chamado *Putty* [16] para facilitar a mesma. O programa *Putty* [16] permite, sem qualquer alteração nas suas

definições, ligação a uma consola que se encontra a rodar no servidor externo. Como o algoritmo desenvolvido não tem qualquer característica gráfica, foi bastante simples fazer a ligação, enviar o algoritmo e realizar todos os testes.

O servidor remoto continha um processador AMD Epyc com quatro núcleos e32 Gigabytes de memória principal, que são essenciais para o algoritmo, uma vez que todo ele funciona em memória primária.

6.2 Características dos *data sets*

Para os testes foram utilizados dois tipos de *data sets*, distinguidos pela sua origem: *data sets* artificiais e *data sets* reais.

Data sets artificiais foram criados utilizando um gerador dos mesmos que permite controlar características como o número de registos, número de dimensões, cardinalidade das dimensões e *skew* do *data set*. Estes foram utilizados para poder testar o comportamento do algoritmo e estrutura realizada, comparando o comportamento do mesmo com o comportamento do algoritmo de Frag-Cubing realizado.

Três *Data sets* reais foram utilizados para demonstrar o comportamento do algoritmo no mundo real, na qual as características das dimensões podem diferenciar grandemente dentro do mesmo *data set*. Este processo é especialmente útil para poder demonstrar que o algoritmo funciona em situações reais, aumentando a sua credibilidade.

Como foi acima indicado, os *data sets* contêm 5 características manipuláveis: o número de registos, o número de dimensões, a cardinalidade dimensional e a *skew* do *data set*.

O número de dimensões é o número de atributos que um registo contêm, uma vez que cada dimensão contem um atributo.

A cardinalidade de cada dimensão é, devido à forma de funcionamento do algoritmo, o maior valor da dimensão. Note-se que o gerador de *data sets* utilizado garante que todos os atributos são utilizados, pelo menos, uma única vez, assumindo também que existem registos suficientes para que sejam utilizados, sem colocar em causa a *skew* dimensional.

Por fim, a *skew* é a dispersão os atributos, isto é, a tendência que os registos têm para conter determinado atributo. Uma *skew* de zero faz com que os atributos tenham uma distribuição uniforme ao longo do *data set*. Aumentar a *skew* do *data set* faz com que a distribuição uniforme se transforme numa distribuição normal, com centro em um valor aleatório determinado pelo algoritmo gerador.

Como parte do objetivo é também perceber se a compressão de TIDs em listas invertidas é uma possível solução para o problema de "*big data*", as características dadas

aos *data* sets tinham valores maiores do que aqueles que são encontrados na maioria dos *data sets* reais, com a exceção da *skew*, que é a única característica que não afeta o tamanho de um data set, mas sim a distribuição dos atributos ao longo do mesmo.

Inicialmente foram utilizados *data sets* artificiais com 10, 30, 50, 70 e 90 milhões de registos, em que cada registo podia ter 30, 50 e 63 dimensões, sendo que as dimensões foram testadas com as cardinalidades de 2500, 5000, 7500 e 10000 e, finalmente, as *skews* testadas foram: 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 e 5.

É necessário justificar que, o valor de 63 dimensões foi um limite que o programa utilizado para gerar os *data sets* artificiais teve, uma vez que não foi possível aumentar o número de dimensões para além desse valor.

Deve notar-se também que os testes realizados faziam alteração de apenas uma das características de cada vez nos data sets, sendo assim, existiram valores que eram usados por predefinição, sendo os mesmos o valor de 10 milhões de registos, 30 dimensões, cardinalidade de 2500 e skew de 1.5.

O valor por predefinição para a *skew* de 1.5 foi escolhido por ser o menor valor testado que permite ao algoritmo CiiCube realizar número de compressões significativo. Para o caso específico dos testes de variação de *skew*, os resultados obtidos foram considerados especialmente interessantes, uma vez que a *skew* é o fator com maior relevância relativamente ao sucesso do algoritmo criado, como será discutido futuramente. Dessa forma, foram utilizados *data sets* com 130 milhões de registos, 30 dimensões e cardinalidade dimensional de 2500. O valor de 130 milhões provém do facto de que o sistema usado não conseguia indexar *data sets* com mais registos.

6.3 Tipos de testes realizados

Como para se conseguir compreender os resultados obtidos nos testes realizados é necessário ter-se uma referência, todos os testes foram realizados em ambos os algoritmos CiiCube e Frag-Cubing implementados.

Os testes realizados podem ser divididos em três tipos: testes de indexação, *point queries* e *subcube queries*.

Testes de indexação são testes realizados com o objetivo de compreender o tempo de indexação dos algoritmos e o tamanho da estrutura criada por ambos. Este teste utiliza duas métricas de medição distintas:

- Tamanho da estrutura: Esta métrica trata de registar o tamanho da memória da estrutura final criada após a indexação do data set. Deve notar-se que esta métrica não é necessariamente a quantidade máxima de memória utilizada durante a

indexação, uma vez que o processo de indexação termina com a eliminação da memória alocada inutilizada;

- Tempo de indexação: Esta métrica visa registar o tempo utilizado pelos algoritmos para transformar o *data set* na estrutura final do cubo de dados.

Testes com *point queries* são utilizados para se poder obter resultados que são apenas influenciados pela velocidade dos algoritmos de interseção. O único valor retirado destes testes é o seu tempo de execução que, quanto maior, mais lento é o algoritmo de interseção.

Testes com *subcube queries* são utilizados para se simular um estudo mais próximo do realizado em ambientes de trabalho reais. Estes são testes múltiplas vezes mais complexos do que testes com *point queries*, pelo que quaisquer diferenças de desempenho nos algoritmos de interseção tornam-se ainda mais evidentes. Para a análise de resultados nestes testes são utilizadas duas métricas:

- Consumo de memória: Esta métrica visa registar a quantidade de memória utilizada para se poder responder à *query* realizada. O resultado obtido conta com o tamanho da estrutura inicial, o tamanho do subcubo criado e o tamanho do array que regista os valores obtidos da análise do subcubo.
- Tempo de processamento: Esta métrica é utilizada para registar o tempo utilizado pelo algoritmo para poder responder à *query* realizada. Note-se que o algoritmo tem a opção de mostrar o resultado da *query* realizada. Como mostrar o resultado é uma operação lenta e não tem interesse para a pesquisa realizada, o resultado das pesquisas não foi mostrado.

Os testes aqui indicados permitem a que sejam retiradas conclusões quanto ao desempenho de qualquer característica fundamental sobre o algoritmo CiiCube.

6.4 Avaliação experimental dos *data sets*

Aqui serão indicados os testes e resultados realizados sobre data sets artificiais e reais.

No sentido de facilitar a descrição dos *data set*s, para o resto deste subcapítulo, o caráter T servirá para indicar o número de registos do *data* set, o carater D servirá para indicar o número de dimensões do *data set*, o carater C servirá para indicar a cardinalidade do *data set* e o carater S servirá para indicar a *skew* do *data set*.

6.4.1 *Data sets* artificiais

De forma a melhor compreender o comportamento do algoritmo CiiCube num ambiente controlado foram realizados testes sobre *data sets* artificiais. Esses testes incluem testes de indexação, *point queries* e *subcube queries*.

6.4.1.1 Testes de indexação

Na *Figura 19* podem ver-se os resultados dos tempos de indexação de *data sets* conforme a variação da cardinalidade. O algoritmo de CiiCube tem um tempo de indexação ligeiramente superior a Frag-Cubing, sendo que o comportamento de ambos os algoritmos é similar, tendo ambos um aumento linear no tempo de processamento conforme o aumento da cardinalidade.

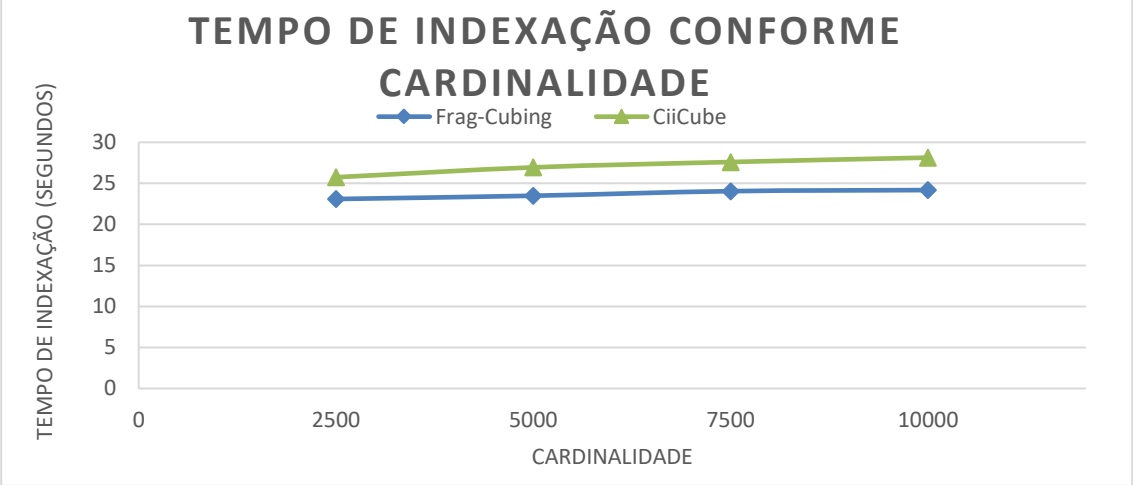


Figura 19. *Tempo de Indexação, em Segundos, Variando a Cardinalidade do Data Set com T = 10M, D = 30, S = 1.5 e C = 2500, 5000, 7500 e 10000*

Resultados para o tempo de indexação relativamente à variação do número de registos podem ser vistos na *Figura 20*. Nessa figura é possível verificar que ambos Frag-Cubing e CiiCube têm um aumento linear do tempo de indexação conforme o aumento do número de registos. Note-se que o tempo de indexação de CiiCube é sempre ligeiramente supeior ao de Frag-Cubing, o que é esperado devido ao processamento extra para realizar a compressão.

Figura 20. *Tempo de Indexação, em Segundos, Variando o Número de Registos do Data Set com D = 30, C = 2500, S = 1.5 e T = 10M, 30M, 50M, 70M e 90M*

Olhando agora para a *Figura 21*, que mostra a variação do tempo de indexação de Frag-Cubing e CiiCube conforme a variação do número de dimensões. Como é possível ver, ambos os algoritmos têm um comportamento similar: o aumento do tempo de indexação aumenta linearmente com o número de dimensões. Mais uma vez, é também possível verificar que Frag-Cubing é ligeiramente mais rápido do que CiiCube para realizar o processo de indexação, o que é esperado devido ao processamento extra para realizar a compressão.

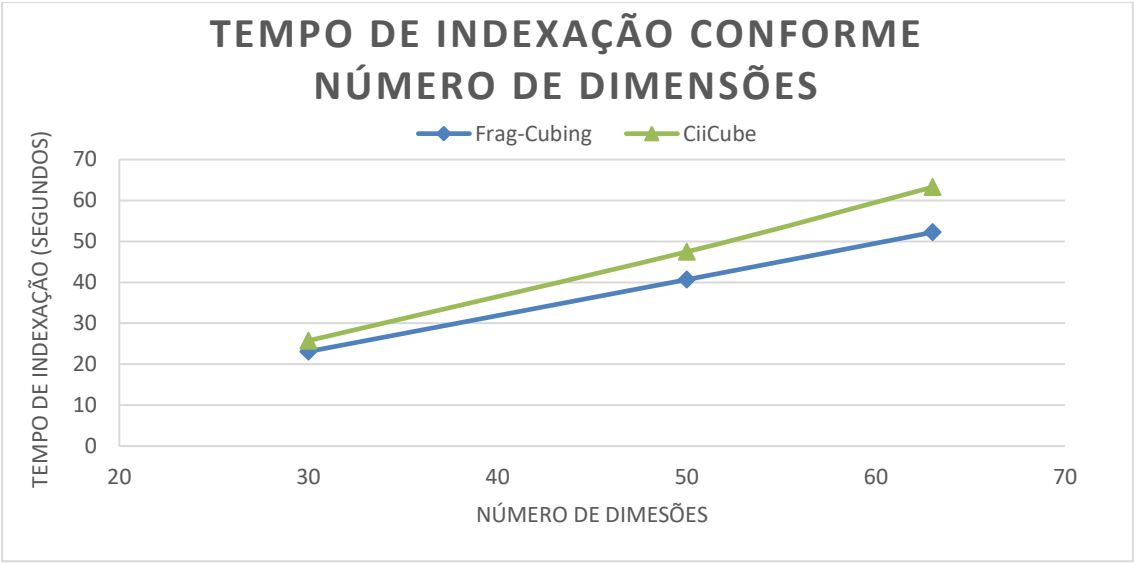


Figura 21. *Tempo de Indexação, em Segundos, Variando o Número de Dimensões do Data Set com T = 10M, C = 2500, S = 1.5 e D = 30, 50 e 63*

Ao contrário dos testes acima indicados, variar o *skew* do *data set* foi o único teste em que, à medida que a *skew* aumenta, é possível ver que o tempo de indexação do algoritmo CiiCube aproxima-se, e consegue mesmo ser inferior, ao tempo de indexação do algoritmo de Frag-Cubing, como pode ser visto na *Figura 22*.

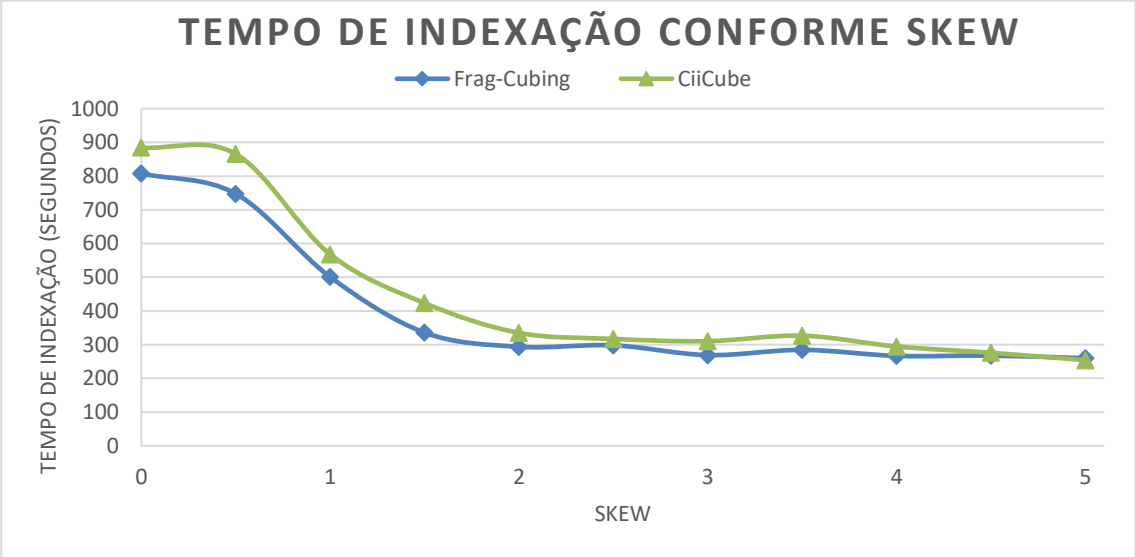


Figura 22. *Tempo de Indexação, em Segundos, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500 e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5*

Observando a *Figura 22*, é possível verificar que existe uma tendência geral para a diminuição do tempo de indexação em ambos os algoritmos. Este comportamento generalizado é resultado da diminuição do número de vezes que é necessário aumentar o tamanho dos *arrays* usados para guardar as listas invertidas de TIDs, que é uma operação lenta conforme indicado no final do subcapítulo 5.2.5.

Relativamente ao tamanho da estrutura indexada em memória, como se pode ver na *Figura 23*, o aumento da cardinalidade não afetou consideravelmente o consumo de memória de nenhum dos programas, o que é algo esperado. Note-se que CiiCube utilizou ligeiramente menos memória do que Frag-Cubing para guardar a estrutura indexada.

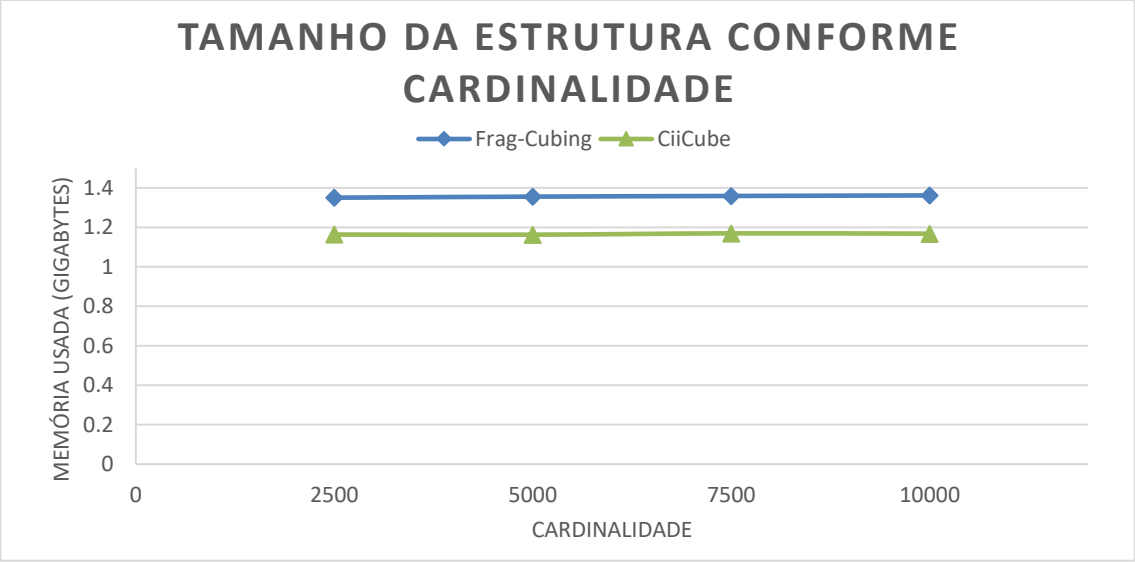


Figura 23. *Tamanho da Estrutura, em Gigabytes, Variando a Cardinalidade do data set com T = 10M, D = 30, S = 1.5 e C = 2500, 5000, 7500 e 10000*

Relativamente ao número de registos a *Figura 24* mostra os valores do consumo de memória conforme a variação do número de registos. Ambos os algoritmos têm um aumento no consumo de memória aproximadamente linear com o aumento do número de registos. Deve notar-se quando o número de registos foi igual a 50M CiiCube utilizou ligeiramente mais memória do que Frag-Cubing, resultado obtido pelo facto do *data* set gerado com 50M de registos possuir uma distribuição de registos que não permite uma eficiente compressão.

Figura 24. *Tamanho da Estrutura, em Gigabytes, Variando o Número de Registos do Data Set com D = 30, C = 2500, S = 1.5 e T = 10M, 30M, 60M e 90M*

Na *Figura 25* é possível observar o consumo de memória para a criação da estrutura em ambos os algoritmos conforme a variação do número de dimensões. Conforme é possível ver, para ambos os algoritmos, existiu um aumento linear do consumo de memória conforme

o aumento do número de dimensões. Deve, mais uma vez, notar-se que o algoritmo CiiCube teve um menor consumo de memória do que Frag-Cubing, o que é resultado da compressão realizada.

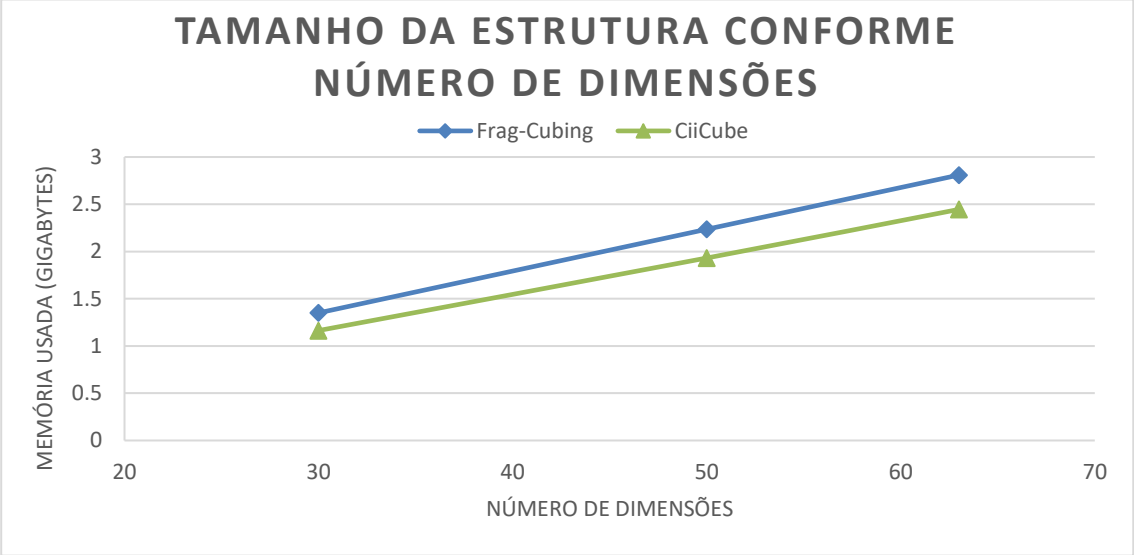


Figura 25. *Tamanho da Estrutura, em Gigabytes, Variando o Número de Dimensões do Data Set com T = 10M, C = 2500, S = 1.5 e D = 30, 50 e 63*

Mais uma vez, assim como com o tempo de indexação, a variação da *skew* provocou diferentes comportamentos para ambos os algoritmos quanto ao tamanho da estrutura finalizada, como pode ser visto na *Figura 26*.

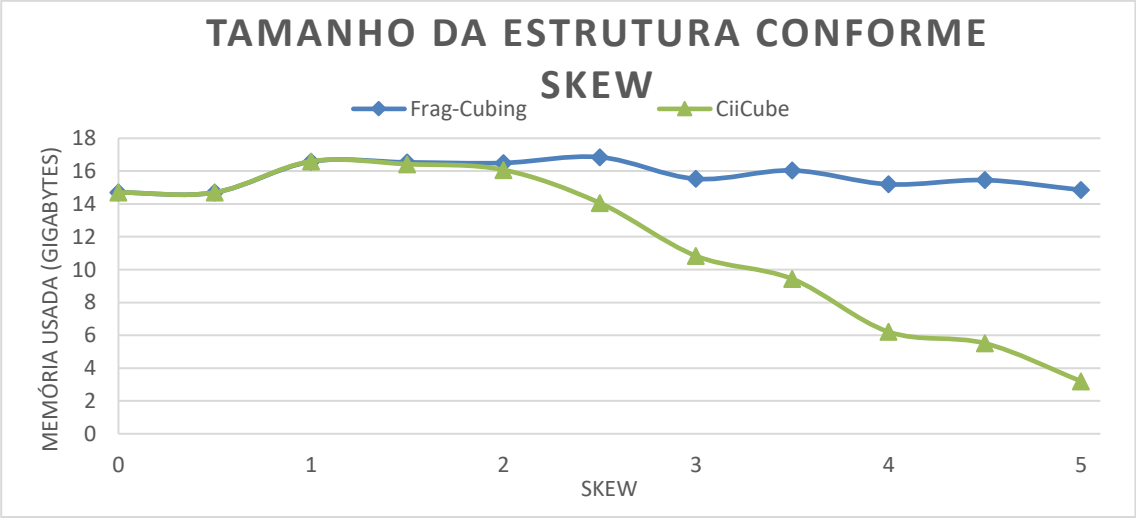


Figura 26. *Tamanho da Estrutura, em Gigabytes, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500 e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5*

Analisando os valores obtidos na *Figura 26*, é possível ver que o algoritmo CiiCube só conseguiu realizar uma variação de memória considerável quando S >= 2. O aumento da efetividade da compressão à medida que a *skew* aumenta está relacionado com o facto de

que, aumentar a *skew* de um *data set* aumenta também as hipóteses de um determinado atributo estar várias vezes repetido em registos seguidos, permitindo a sua compressão.

É possível verificar também que, no pior caso, o algoritmo CiiCube utiliza aproximadamente a mesma quantidade de memória que o algoritmo de Frag-Cubing, como pode ser visto quando S = 1. Isto indica que, baseado apenas no tamanho da estrutura em memória, o algoritmo de CiiCube é sempre uma opção superior ao algoritmo de Frag-Cubing, uma vez que, no pior caso, utiliza a mesma quantidade de memória para a estrutura.

Um resultado notável que permite comprovar a eficiência de CiiCube em *data sets* com uma *skew* elevada foi um teste de indexação com T = 500 milhões, D = 30, C = 2500 e S = 5. Nesse teste o algoritmo CiiCube teve um tempo de indexação de 16,40 minutos e a estrutura indexada ocupou aproximadamente 11.99 gigabytes na memória principal. O algoritmo de Frag-Cubing não conseguiu indexar o mesmo data set com o mesmo sistema, que contém 30 gigabytes de memória.

6.4.1.2 Testes de *point queries*

Foram realizados testes de point query com o objetivo de comparar os comportamentos dos algoritmos de Frag-Cubing e CiiCube. Nestes testes, foram utilizados *data sets* com T = 130 milhões, D = 30, C = 2500 e com 30 operadores *equal*. Os *data sets* utilizados tiveram os seguintes valores de *skew*: 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 e 5.

Deve notar-se que foram apenas realizados este tipo de testes uma vez que, baseado nos testes de indexação, a *skew* é a única variável que aparenta afetar diretamente o tamanho dos intervalos de compressão criados, que por sua vez afetam o desempenho do algoritmo de interceção, que é onde a maior parte do processamento de *point queries* é realizado.

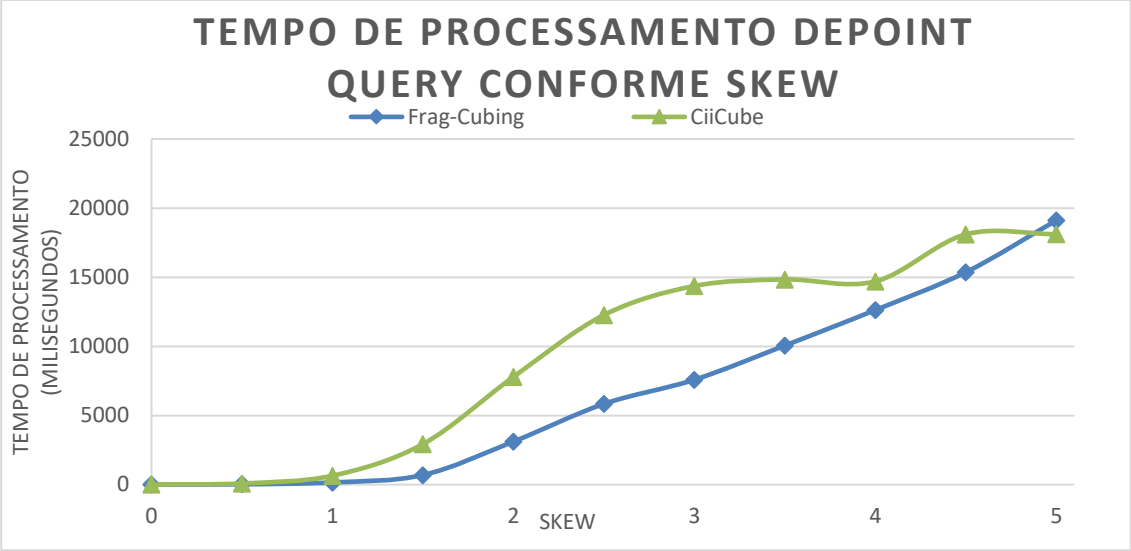


Figura 27. *Tempo de Processamento de Point Queries Variando a Skew em Data Sets com T = 130M, D = 30, C = 2500, 30 Operadores Equal e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5.*

Como é possível verificar através da *Figura 27*, em geral, CiiCube tem um tempo de processamento de *point queries* superior a Frag-Cubing. Este resultado é obtido pelo maior processamento existente no algoritmo de interseção de CiiCube, quando comparado com o processamento existente no algoritmo de interseção de Frag-Cubing.

Outro resultado interessante é que, em casos onde quase nenhuma compressão é realizada, como é o caso de quando S = 0 e S = 0.5, os tempos de processamento de ambos os algoritmos são similares, o que também é esperado quando se analisa o algoritmo de interseção. Deve adicionar-se que, quando S = 0 ou S = 0.5, a compressão realizada é quase nula pois os valores são distribuídos de uma forma aproximadamente uniforme, o que faz com que seja raro que 3 ou mais registos seguidos tenham o mesmo valor de atributo.

Um resultado surpreendente é que, quando S = 5, o tempo de processamento de Frag-Cubing foi superior ao tempo de processamento de CiiCube. Este valor obtido indica que, quando os intervalos comprimidos são suficientemente grandes, é possível aumentar o desempenho, uma vez que o algoritmo de interseção de CiiCube consegue processar os intervalos de uma forma eficiente.

Finalmente, é possível verificar que quando S > 2, existe um crescimento aproximadamente linear quanto ao tempo para responder a *queries*, o algoritmo CiiCube teve também um crescimento, porém mais inconsistente. Notavelmente, quando S = 3, 3.5 e 4, o tempo de processamento manteve-se idêntico. Este comportamento do algoritmo CiiCube é um resultado direto do sucesso da compressão realizada.

6.4.1.3 Testes de *subcube queries*

Relativamente à métrica do tempo de processamento de *subcube queries*, a *Figura 28* mostra o os resultados de testes dessa métrica conforme a variação da cardinalidade.

Como é possível verificar na *Figura 28*, ambos os algoritmos tiveram um aumento linear do tempo de processamento conforme o aumento da cardinalidade. Este resultado é esperado pois aumentar a cardinalidade das dimensões inquiridas obriga a um aumento do número de diferentes point *queries realizadas* durante o a *subcube query*. Para além disso, olhando para os resultados, CiiCube é consideravelmente mais lento que Frag-Cubing a realizar as *subcube queries*. Este resultado é esperado pelo facto de que, para uma *skew* de 1.5, a operação de *point query* ser consideravelmente mais lenta, como visto no subcapítulo 6.4.1.2.

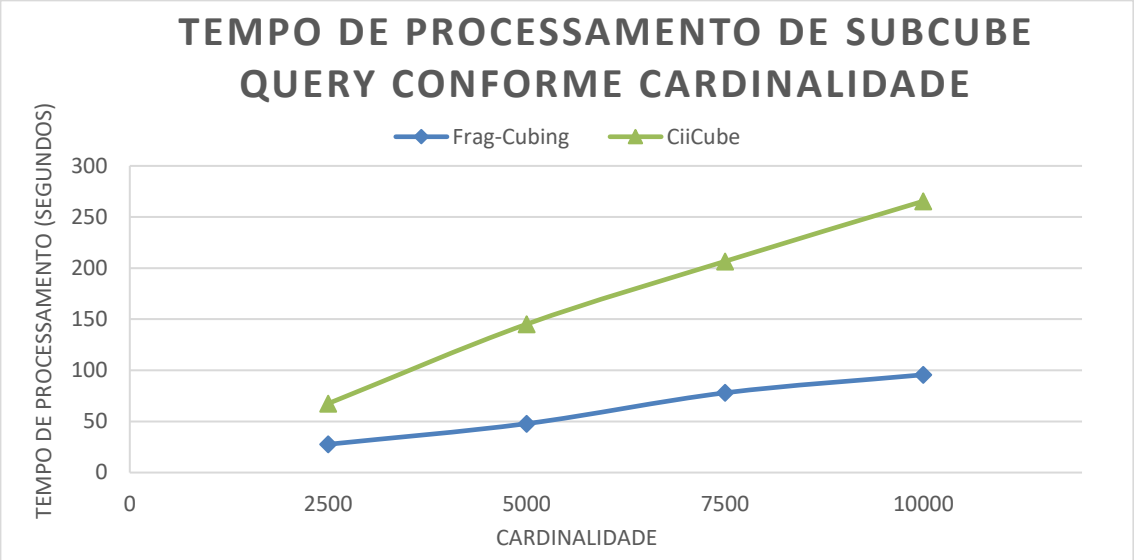


Figura 28. *Tempo de Processamento DE Subcube Queries, em Segundos, Variando a Cardinalidade do Data Set com T = 10M, D = 30, S = 1.5, 1 Operadores Equal e 2 Operadores Inquire e C = 2500, 5000, 7500 e 10000.*

A *Figura 29* apresenta o tempo de processamento de *subcube queries* variando o número de registos. É possível observar nessa figura que, ainda que exista uma tendência para o aumento do tempo de processamento conforme o aumento do número de registos, aumentar essa tendência não é algo que acontece sempre. O facto de CiiCube ser mais rápido a realizar a operação com 70M de registos do que com 50M de registos é indicativo de uma melhor compressão com o primeiro *data set*. Note-se que a métrica de tamanho da estrutura em testes com variação do número de registos, realizada na subsecção 6.4.1.1, indicou que CiiCube teve dificuldade em realizar compressão com o *data set* utilizado com 50M de registos. Como era de esperar, Frag-Cubing realizou a *subcube query* em menos tempo do que CiiCube, uma vez que a operação de interceção de Frag-Cubing tem um melhor desempenho geral.

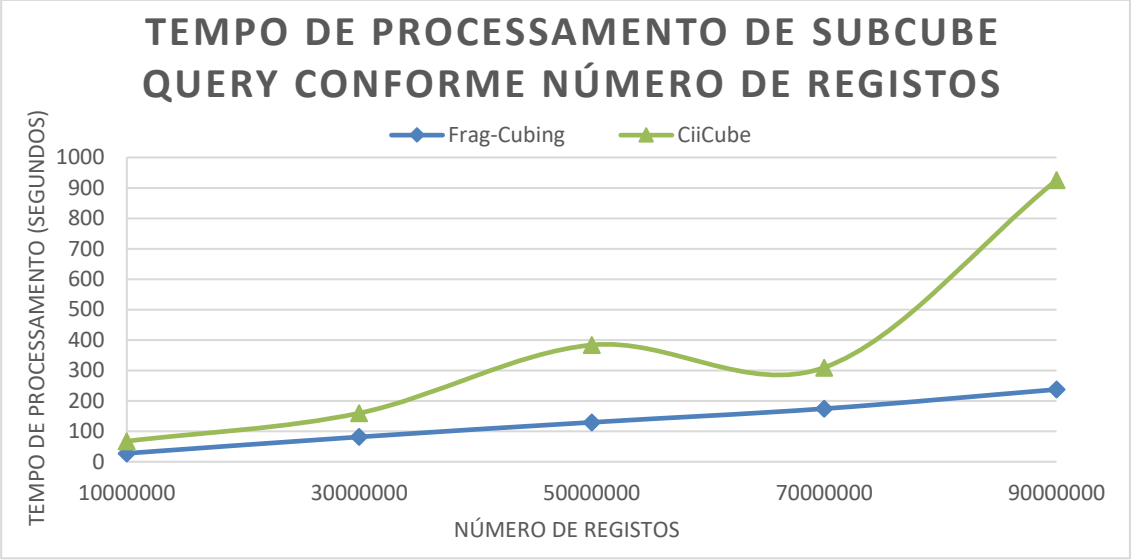


Figura 29. *Tempo de Processamento em Subcube Queries, em Segundos, Variando o Número de Registos do Data Set com D = 30, C = 2500, 1 Operadores Equal, 2 Operadores Inquire e S = 1.5 e T = 10M, 30M, 50M, 70M e 90M.*

Na *Figura 30* é possível ver tempo de processamento de *subcube queries* variando o número de dimensões. Como é possível ver, em Frag-Cubing, a variação do número de dimensões não tende a afetar consideravelmente o tempo de processamento para a operação realizada. Por outro lado, CiiCube teve uma variação considerável quando o número de dimensões passou de 50 para 63. Esta variação abrupta é resultado da compressão existente sobre as dimensões inquiridas nas *queries* de teste realizadas e não um resultado diretamente obtido devido à variação do número de dimensões.

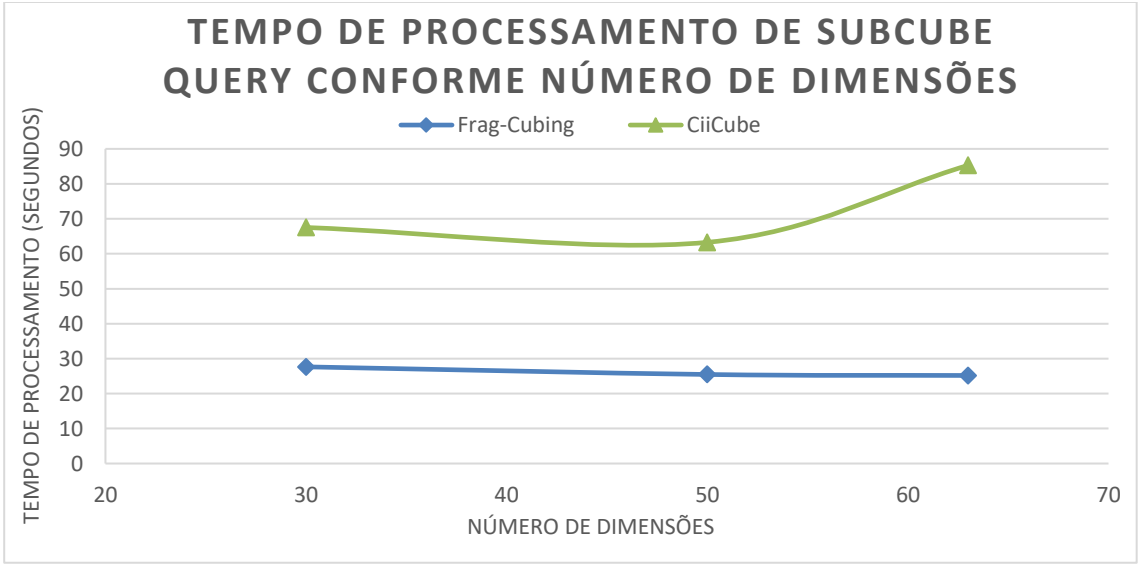


Figura 30. *Tempo de Processamento em Subcube Queries, em Segundos, Variando o Número de Dimensões do Data Set com T = 10M, C = 2500, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e D = 30, 50 e 63.*

Mais uma vez, a variação de *skew* dos *data sets* volta a trazer resultados surpreendente. A *Figura 31* mostra os resultados do tempo de processamento de testes com point queries, usando dois operadores inquire e um operador equal, que instancia o atributo mais recorrente da sua dimensão. Os *data sets* usados têm 130 milhões de registos, 30 dimensões e cardinalidade de 2500.

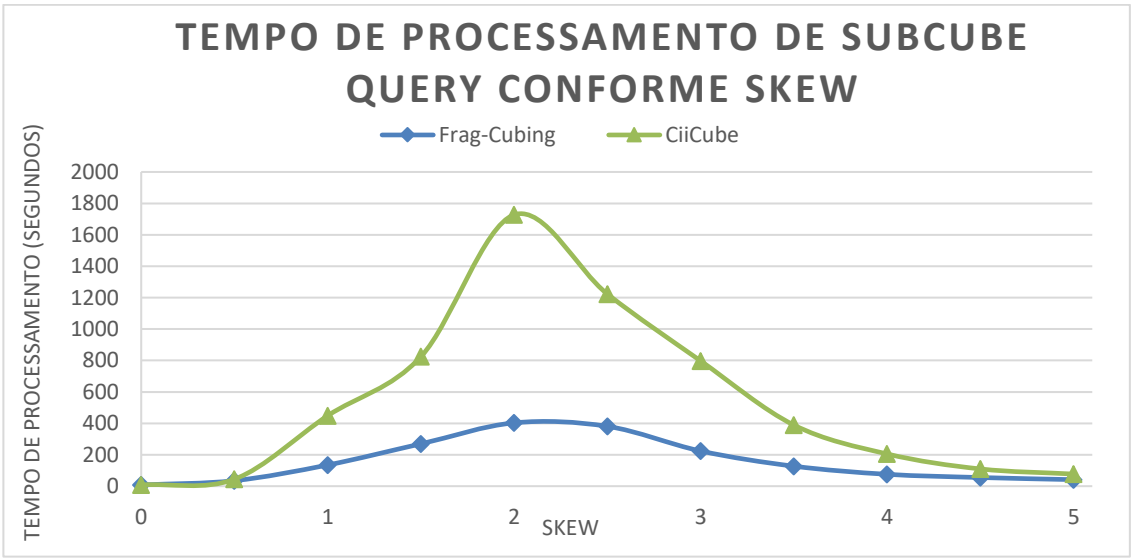


Figura 31. *Tempo de Processamento em Subcube Queries, em Segundos, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500, 1 Operadores Equal, 2 Operadores Inquire e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5.*

Como é possível aferir da *Figura 31*, ambos os algoritmos têm maior tempo de processamento quando S = 2. Quando S = 0 e 0.5 ambos os algoritmos testados tiveram aproximadamente o mesmo tempo de processamento, indicando que, quando a compressão é mínima, ambos os algoritmos têm o mesmo comportamento.

A maior desvantagem do algoritmo CiiCube são os muito superiores tempos de processamento quando S se encontra no intervalo [1; 4]. Este tipo de diferença pode apenas ser atribuída ao algoritmo de interseção. Enquanto que o algoritmo de interseção de CiiCube necessita de escolher, para cada objeto *DIntArray* a ser intercetado, o valor seguinte a usar na interseção, o algoritmo de interseção de Frag-Cubing não necessita de o fazer. Este processo de escolha aparenta ser o maior fator para a diferença nos tempos de processamento das *subcube queries*.

Relativamente ao consumo de memória *subcube queries*, a *Figura 32* mostra que o aumento da cardinalidade resulta num maior consumo de memória para realizar *subcube queries*. Este resultado é esperado pois o número de resultados de *point* queries realizadas durante as *subcube queries* para guardar é maior.

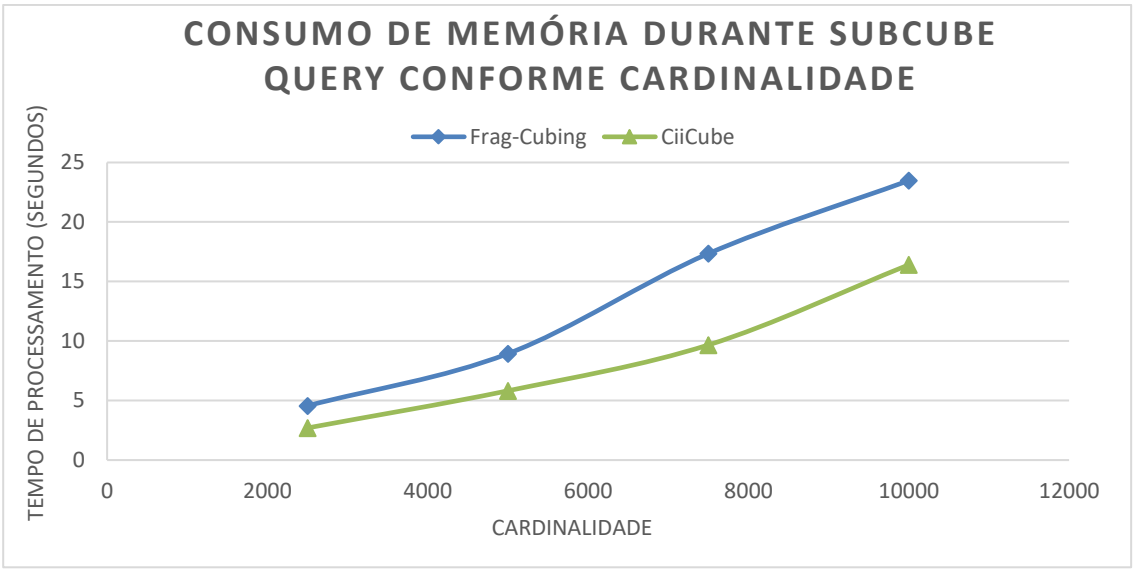


Figura 32. *Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com T = 10M, D = 30, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e C = 2500, 5000, 7500 e 10000*

A *Figura 33* mostra os resultados a testes de *subcube queries* variando o número de registos do *data set*. Como pode ser visto nessa figura, ambos os algoritmos têm um aumento no consumo de memória linear com o aumento do número de registos. Note-se que CiiCube teve um menor consumo de memória, muito do qual veio do tamanho da estrutura após a operação de indexação.

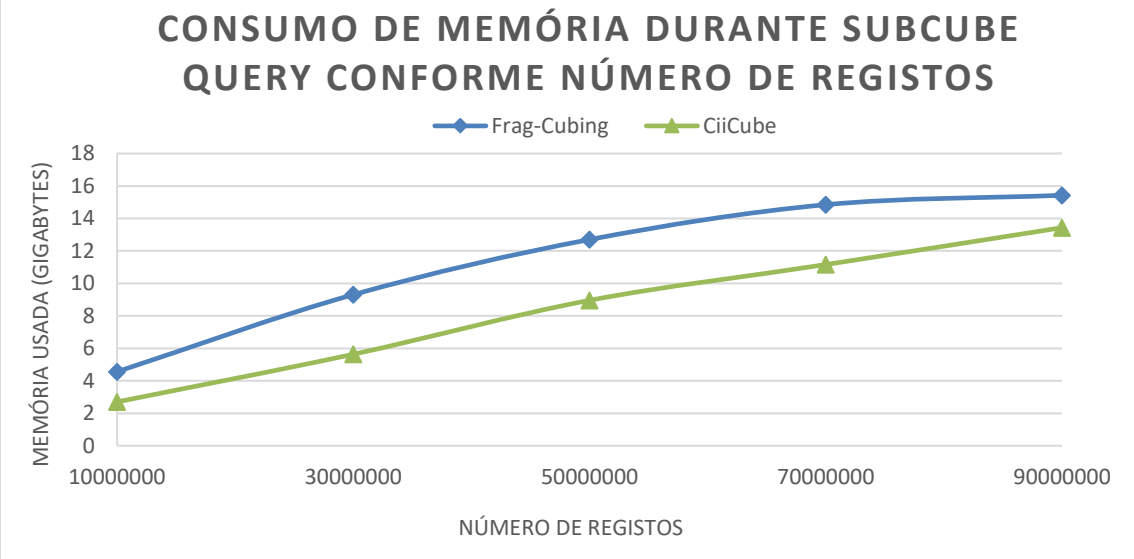


Figura 33. *Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com D = 30, C = 2500, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e T = 10M, 30M, 50M, 70M e 90M*

A *Figura 34* mostra o resultado dos testes de *subcube query* variando o número de dimensões. É possível observar através da figura que a existe um aumento no consumo de memória para a realização de *subcube queries*, esse aumento, porém, não aparenta estar relacionado com a *query* em si, mas sim com o tamanho da estrutura após a operação de

indexação. Quando retirado o tamanho da estrutura, não aparenta existir qualquer relação entre a variação do número de dimensões e a memória utilizada para a realização de *subcube queries*.

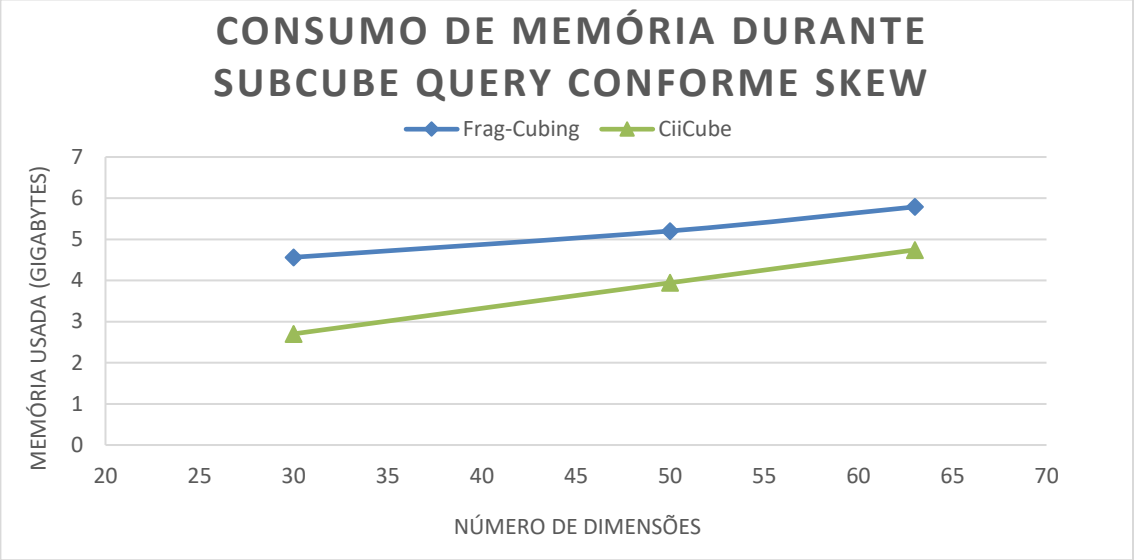


Figura 34. *Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com T = 10M, C = 2500, S = 1.5, 1 Operadores Equal, 2 Operadores Inquire e D = 30, 50 e 63*

A *Figura 35* mostra os resultados do consumo de memória de testes com *point queries* variando a skew dos *data sets*. Como se pode observar, o aumento da *skew* dos *data sets* provoca uma maior diferença na quantidade de memória utilizada para realizar a *query*, porém, a larga maior parte dessa diferença provêm do tamanho da estrutura após indexação.

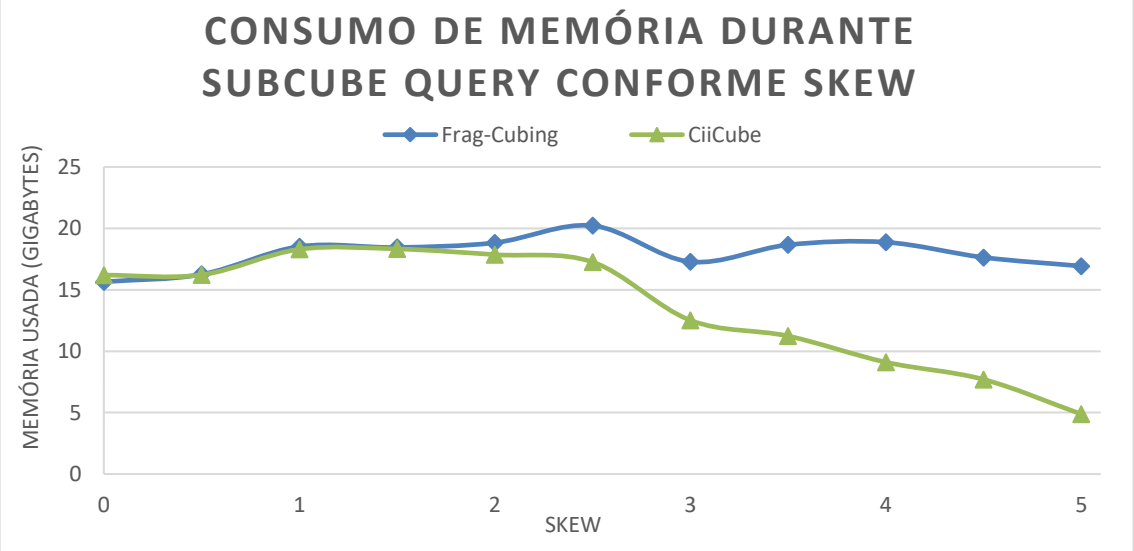


Figura 35. *Consumo de Memória em Subcube Queries, em Gigabytes, Variando a Skew do Data Set com T = 130M, D = 30, C = 2500, 1 Operadores Equal, 2 Operadores Inquire e S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 e 5*

6.4.2 *Data sets* reais

Ainda que seja possível testar múltiplos ambientes diferentes com *data sets* artificiais, nem sempre o desempenho num ambiente controlado é igual ao desempenho no mundo real. Com isto dito foram também realizados testes de indexação e *subcube queries* em *data sets* reais.

6.4.2.1 Data sets utilizados

Foram utilizados 3 *data sets* reais com o objetivo de averiguar o comportamento do algoritmo CiiCube num ambiente real. Para tal, foram utilizados 3 data sets distintos: *Forest Covertype* [17], *Connect-4* [18] e *Exame* [19].

O primeiro data set utilizado foi o "*Forest Covertype Data set*". Este contém 512.012 registos com 54 atributos. A cardinalidades dos atributos são as seguintes: 3858, 361, 67, 1398, 801, 7118, 255, 255, 255, 7174, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 e 7.

O segundo data set utilizado foi o "Connect-4". Este contém 67.557 registos com 42 atributos. Relativamente às cardinalidades, uma das dimensões tem cardinalidade 4 enquanto que as restantes têm cardinalidade 3.

O terceiro data set utilizado foi o "Exames". Este contém 26.651.926 registos de exames de Covid-19 realizados em diferentes hospitais do Brasil. Cada registo deste data set tem 9 atributos diferentes, sendo que os atributos têm as seguintes cardinalidades: 562943, 975188, 425, 97, 1986, 2355, 63778, 105 e 2064.

6.4.2.2 *Data set Forest Covertype*

Como pode ser visto na *Figura 36*, o tempo de indexação médio para o data set *Forest Covertype* foi, para o algoritmo CiiCube, de 1779 milissegundos, enquanto que Frag-Cubing necessitou de 1976 milissegundos. O tamanho da estrutura final em CiiCube foi de 42.7 megabytes e, em Frag-Cubing, de 143 megabytes.

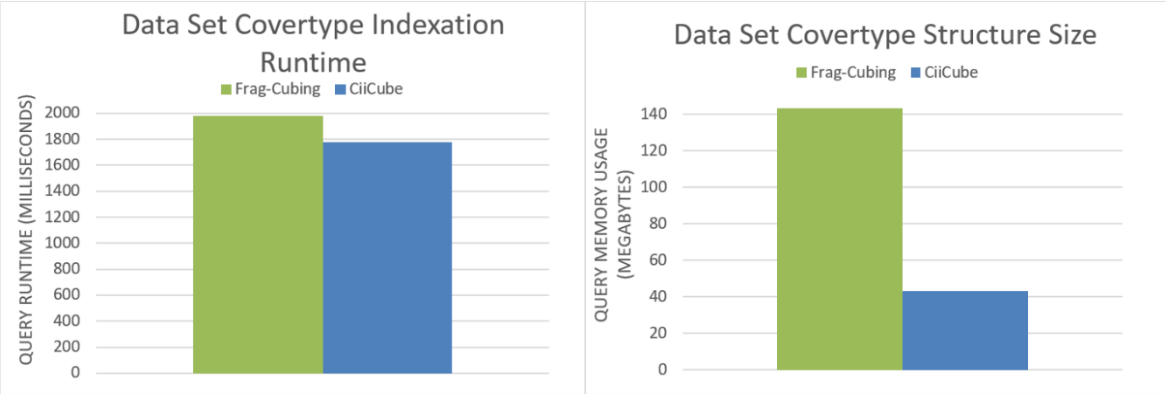


Figura 36. *Tempo de Indexação e Tamanho da Estrutura para o Data Set Forest Covtype*

O tempo de indexação mais rápido para o algoritmo CiiCube é resultado direto da menor necessidade de realizar a operação de aumentar o tamanho dos arrays que guardam os TIDs, como é explicado no capítulo 4.2.5.

É também possível ver que o algoritmo CiiCube utilizou cerca de 25% da memória que Frag-Cubing utilizou para manter a estrutura em memória. Este valor é resultado da compressão realizada sobre as dimensões com cardinalidade binária, que permitem que o algoritmo de compressão consiga ser bastante efetivo.

Para além de estes de indexação, foram realizadas três *subcube queries* sobre o *data set Forest CoverType.*

A primeira *subcube query* realizada teve três operadores *inquire* sobre as dimensões com cardinalidade 2, 2 e 7, respetivamente, sendo o input colocado em ambos os algoritmos de “q * ? ? ?”. Como pode ser visto na *Figura 37*, o algoritmo de Frag-Cubing processou a query em 152 milissegundos e utilizou 173.2 megabytes, enquanto CiiCube utilizou 179 milissegundos e utilizou 64.6 megabytes para realizar a mesma operação.

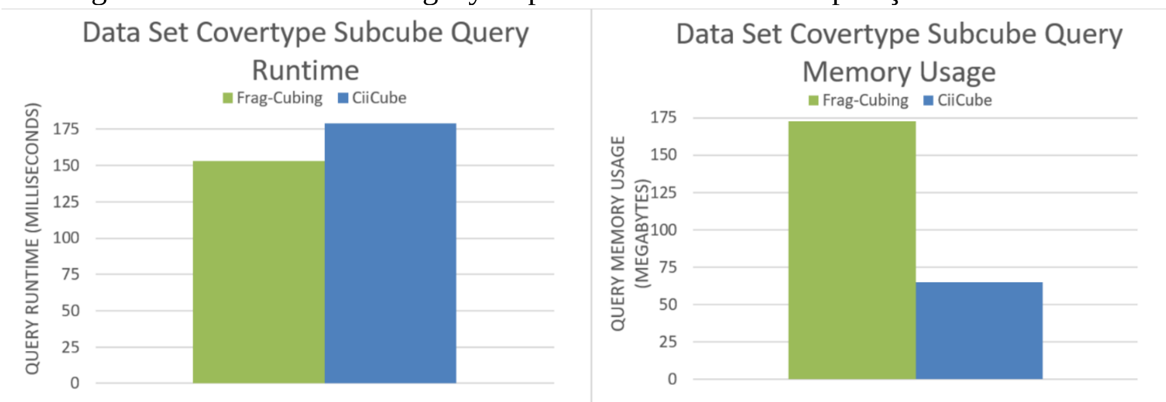


Figura 37. *Tempo de processamento e memória utilizada para uma subcube query com três operadores inquire sobre as dimensões com cardinalidade 2, 2 e 7, pertencentes ao data set Forest Covertype*

A segunda subcube query realizada teve também três operadores inquire, desta vez sobre dimensões com cadinhadas de 3858, 1398 e 801, respetivamente, sendo o input

colocado em ambos os algoritmos de “q * * * ? * ? * * * ? *”. Como pode ser visto na *Figura* 38, Frag-Cubing necessitou de 22 segundos e utilizou 8.53 gigabytes para responder, enquanto que CiiCube necessitou de 35 segundos e consumiu 8.21 gigabytes de memória.

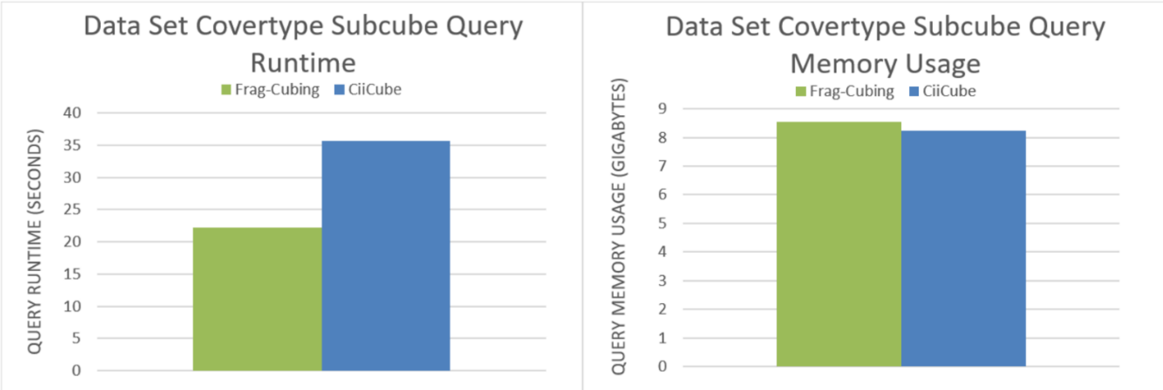


Figura 38. *Tempo de processamento e memória utilizada para uma subcube query com três operadores inquire sobre as dimensões com cardinalidade 3858, 1398 e 801, pertencentes ao data set Forest Covertype*

A terceira *subcube query* realizada teve quatro operadores *inquire* sobre as dimensões com cardinalidade de 3858, 1398, 2 e 2, respetivamente, , sendo o *input* colocado em ambos os algoritmos de “q ? * * ? * ? ? *”. Como pode ser visto na *Figura* 39, Frag-Cubing necessitou de 190.6 segundos e 12.15 gigabytes para responder a esta *query*. Por outro lado, CiiCube utilizou 156.1 segundos e 11.99 gigabytes para realizar a mesma *query*.

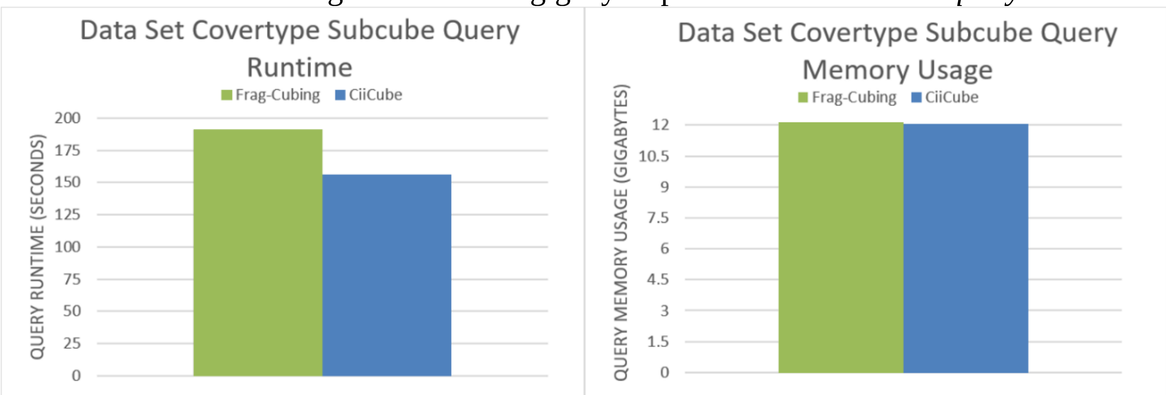


Figura 39. *Tempo de processamento e memória utilizada para uma subcube query com quatro operadores inquire sobre as dimensões com cardinalidade 3858, 801, 2 e 2, pertencentes ao data set Forest Covertype*

A primeira *query* realizada era consideravelmente pequena, uma vez que o número de pesquisas realizadas sobre o subcubo nessa query foi 72, um valor pequeno relativamente às *queries* realizadas neste estudo. Ainda assim, é possível concluir que Frag-Cubing tem um ligeiramente melhor tempo de processamento para relações pequenas.

Na segunda query realizada, CiiCube necessitou de quase do dobro do tempo de processamento que Frag-Cubing utilizou. Esta *query* mostra uma situação onde CiiCube tem um tempo de processamento muito superior ao tempo de processamento de Frag-Cubing. A causa dessa diferença é, muito provavelmente, os baixos níveis de compressão existentes nas dimensões inquiridas, sendo este um caso simular aos testes realizados na *Figura 31*,

quando a *skew* tem o valor de 2. Em ambos os algoritmos, o tamanho necessário para realizar a *query* foi várias vezes superior ao tamanho da estrutura indexada, o que é resultado da quantidade massiva de *point queries* realizadas e guardadas em memória durante a *subcube query* realizada.

A terceira *query* realizada junta duas dimensões com cardinalidade alta e duas dimensões com baixa cardinalidade. Nesta *query*, CiiCube foi cerca de 20% mais rápido do que Frag-Cubing, resultado causado pela vantagem elevada que o algoritmo de CiiCube tem na interseção das duas dimensões com baixa cardinalidade de alta *skew*. Mais uma vez, em ambos os algoritmos, o tamanho necessário para realizar a *query* foi várias vezes superior ao tamanho da estrutura indexada, o que é resultado da quantidade massiva de *point queries* realizadas e guardadas em memória durante a *subcube query* realizada.

6.4.2.3 *Data set Connect-4*

Tal como pode ser visto na *Figura* 40, testes realizados utilizando o *data set Connect-4* mostram que CiiCube necessitou de 274.67 milissegundos e 26.44 megabytes para indexar esse *data set*, enquanto Frag-Cubing necessitou de 269.67 milissegundos e 35.23 megabytes para realizar a mesma operação.

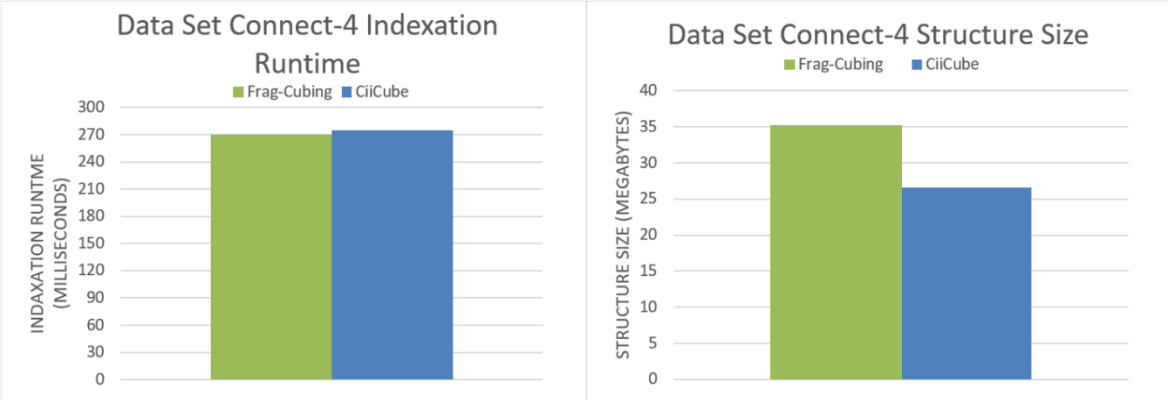


Figura 40. *Tempo de Indexação e Tamanho da Estrutura final para o Data Set Connect-4*

Ainda que este *data set* tenha um pequeno número de registos e baixa cardinalidade, o algoritmo CiiCube conseguiu utilizar menos 25% de memória relativamente a Frag-Cubing. Esta tão considerável diferença num data set tão pequeno é resultado do elevado *skew* existente em algumas das dimensões do mesmo.

Foi realizada sobre este *data set* uma única *subcube query*, que contém o operador *equal*, realizado sobre a dimensão com cardinalidade 4, instanciado o valor mais comum dessa dimensão, e 10 operadores inquire sobre outras dimensões com cardinalidade 3. A query usada para teste foi "q 2 ? ? ? ? ? ? ? ? ? ? *". Como pode ser visto na *Figura 41*, Frag-Cubing utilizou 4.75 segundos e 891.85 megabytes para realizar a *query*, enquanto CiiCube utilizou 8.87 segundos e 825.67 megabytes para realizar a mesma operação.

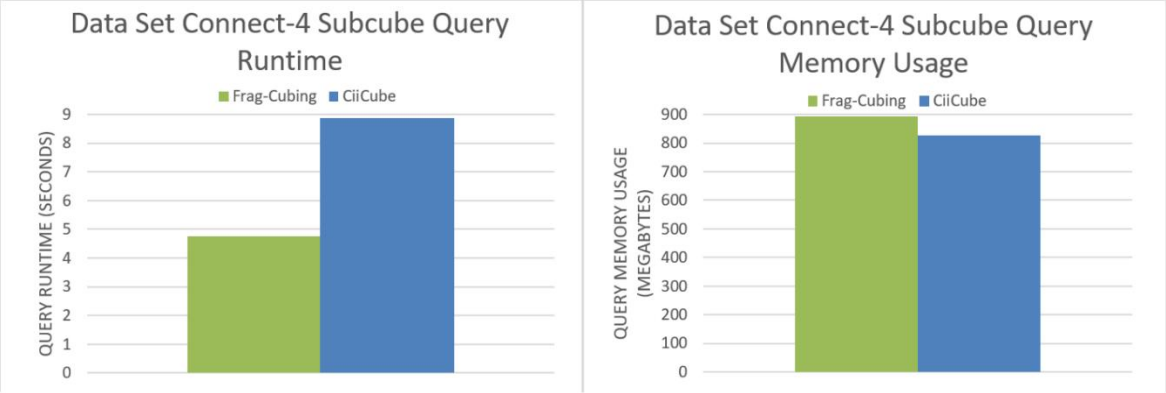


Figura 41. *Tempo de processamento e utilização de memória para subcube queries com 1 operador equal e 10 operadores inquire sobre o data set Connect-4*

A diferença no consumo de memória para a realização das *queries* vem, maioritariamente, da diferença no tamanho da estrutura inicial, o que indica que a compressão não é muito necessária durante *subcube queries*.

6.4.2.4 Data set Exame

Como é visível na *Figura 42*, a indexação do *data set Exame* foi realizada em 16.4 segundos e utilizou 1037 megabytes, para Frag-Cubing. O algoritmo CiiCube, por outro lado, necessitou de 17.7 segundos e a estrutura final ocupava 616 megabytes de memória.

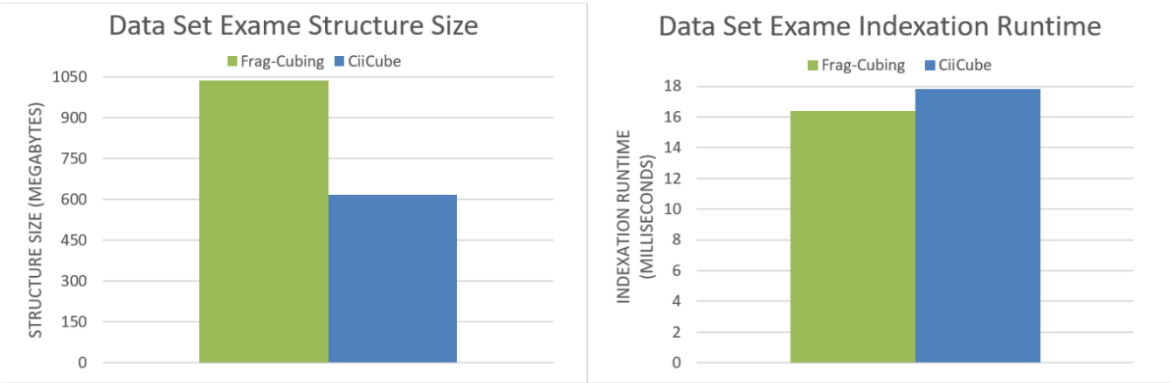


Figura 42. *Tempo de Indexação e Tamanho da Estrutura Final para o Data Set Exame*

Neste caso, CiiCube conseguiu reduzir a memória necessária para cerca de metade, consumindo cerca de metade da memória que Frag-Cubing utiliza. Este bom desempenho é resultado do tamanho bastante elevado do data set que permite várias oportunidades para que se possa realizar compressão.

Foi também realizada uma *subcube query* com um operador *equal*, composto pelo valor mais comum da dimensão com cardinalidade 97, e dois operadores *inquire* sobre as

dimensões com cardinalidade 1985 e 2355. A query usada para teste foi "`q * * * 1 ? ? * * *`". Olhando para os resultados expostos na *Figura 43*, Frag-Cubing utilizou cerca de 108 segundos e 2.34 gigabytes para responder à *query*, enquanto CiiCube necessitou de cerca de 436 segundos e 1.68 gigabytes para a mesma operação.

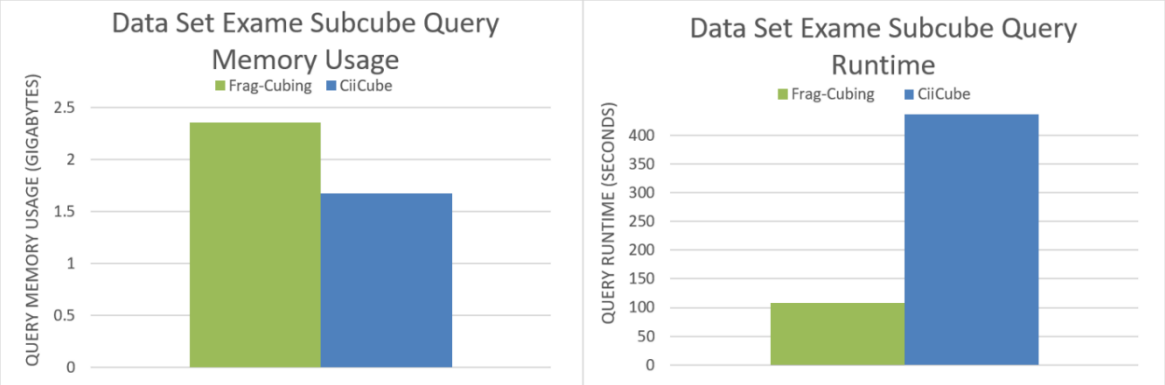


Figura 43. *Tempo de processamento e utilização de memória para subcube queries com 1 operador equal e 2 operadores inquire sobre o data set Exame*

A maior parte da diferença no consumo de memória entre os dois algoritmos é resultado do tamanho da estrutura inicial, sublinhando que a compressão do subcubo de dados realizados durante a subcube query não é fundamental, ainda que permita consumir menos memória. É necessário notar também que o algoritmo CiiCube necessitou de mais de quatro vezes mais tempo para realizar a query do que Frag-Cubing. Isto indica que, ainda que tenha havido sucesso na compressão, os intervalos realizados são relativamente pequenos, o que aumenta consideravelmente o tempo de processamento.

6.5 Conclusões sobre o CiiCube tendo em conta os resultados

CiiCube foi um algoritmo desenhado com o propósito de diminuir a quantidade de memória utilizada para indexar data sets e realizar operações sobre os mesmos, utilizando também tempos de operação compatíveis com Frag-Cubing.

Deve sublinhar-se que todas as conclusões obtidas têm, como ponto de referência o algoritmo de Frag-Cubing, algoritmo esse que está bem colocado na indústria como um dos mais conhecidos, mais utilizados para operações OLAP e com mais de 320 vezes referenciado [20] em estudos como aquele que foi realizado ao longo deste estágio.

Olhando para os resultados obtidos é possível constatar que CiiCube atingiu o objetivo de diminuir o consumo de memória. Todos os testes com *data sets* reais confirmarem que existiu uma diminuição efetiva na memória utilizada para criar e processar cubos de dados.

Olhando também para os resultados obtidos em *data sets* artificiais, é possível verificar que, mesmo no pior caso em que não é possível realizar qualquer quantidade de compressão

que seja notório, CiiCube utiliza aproximadamente a mesma quantidade de memória que Frag-Cubing.

O maior ponto negativo de CiiCube são os tempos de resposta a *queries*. Realizar operações de interseção sobre a listas de TIDs comprimidas, *DIntArray*, tem uma clara penalidade sobre o tempo de indexação, quando comparado com a indexação de listas de TIDs sem qualquer compressão. Foi possível ver, tanto com *data sets* artificiais, como com *data sets* reais, casos em que CiiCube necessita de mais de quatro vezes mais tempo do que Frag-Cubing para processar *subcube queries*.

6.6 Escrita do artigo científico

O último processo realizado durante o estágio foi a escrita de um artigo científico para publicação num jornal científico. Este processo utilizou quase toda a informação descrita neste relatório sobre o algoritmo.

O processo de escrita do artigo foi longo e necessitou de múltiplas revisões por parte de ambos os orientadores. Como o objetivo do artigo era obter a sua publicação num jornal internacional, o artigo foi escrito em inglês. No Anexo B pode ser visto na integra o artigo realizado.

Até ao momento de escrita deste relatório, o artigo encontra-se em processo de revisão por parte de revisores do jornal *Algorithms*.

7 CONCLUSÕES E TRABALHO FUTURO

Este estágio teve como objetivos a conceção e implementação de um algoritmo de cubo de dados, baseado em Frag-Cubing, que implementasse um sistema de compressão com o objetivo de permitir a diminuição do consumo de memória. Para além disso outro objetivo do estágio foi realizar um estudo sobre o comportamento do algoritmo criado e escrever um artigo científico sobre o mesmo.

Para realizar os objetivos propostos existiram múltiplas fases: o estudo dos conceitos base do tema, o estudo de artigos relacionados, a implementação do algoritmo de Frag-Cubing, múltiplas implementações de algoritmos de compressão, testes sobre o algoritmo final e escrita do artigo científico.

Olhando para todo o trabalho realizado e explanado neste relatório, é sólido admitir que todos os objetivos delineados foram cumpridos com sucesso. Foi implementado um algoritmo que consegue, efetivamente, diminuir a quantidade de memória utilizada em cubos de dados, foram realizados testes com o algoritmo implementado e foi escrito um artigo científico sobre o mesmo.

O algoritmo criado, apesar de não ser perfeito, especificamente quanto ao tempo de processamento de algumas operações, é viável para casos onde existe necessidade de reduzir a memória utilizada.

Deve sublinhar-se que o artigo científico realizado encontra-se em processo de revisão por parte de um jornal internacional, sendo esse o motivo de não estar publicado.

Não deve ser esquecido todo o apoio proporcionado pela CISUC durante todo o tempo de estágio, seja na forma do orientador Prof. Rodrigo Rocha que sempre esteve presente para ajudar, ou na forma da obtenção de material que permitisse a realização do trabalho, como foi o caso da obtenção do servidor onde foram realizados os testes.

Olhando para os resultados obtidos em testes, é notório que o maior problema do algoritmo desenvolvido, CiiCube, é o tempo de resposta a *queries*. A maior causa desse tempo é o processamento de listas invertidas comprimidas. Para tentar diminuir o tempo de processamento, no casos de *subcube queries*¸ como trabalho futuro poder-se-ia implementar uma versão em que os subcubos de dados criados durante *subcube queries* não fossem comprimidos, permitindo, dessa forma aumentar o desempenho do algoritmo neste tipo de operações.

REFERÊNCIAS

[1] "História," ISEC, [Online]. Available: https://www.isec.pt/pt/instituto/#lnkHistoria. [Acedido em 11 7 2021].

[2] "Apresentação," ISEC, [Online]. Available: https://www.isec.pt/pt/instituto/#lnkApresentacao. [Acedido em 11 7 2021].

[3] "Factos e Números," ISEC, [Online]. Available: https://www.isec.pt/pt/instituto/#lnkFactosNumeros. [Acedido em 11 7 2021].

[4] "History," CISUC, [Online]. Available: https://www.cisuc.uc.pt/en/history. [Acedido em 11 7 2021].

[5] C. Salley e E. Codd, "Providing OLAP to User-Analysts: An IT Mandate," 1998.

[6] J. Zobel, A. Moffat e K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Transactions on Database Systems,* vol. 23, nº 4, p. 453–490, 1998.

[7] J. Gray, S. Chaudhuri,, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow e H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," *Data Mining and Knowledge Discovery,* p. pages29–53, 1997.

[8] "techopedia," [Online]. Available: https://www.techopedia.com/definition/28530/data-cube. [Acedido em 9 9 2021].

[9] [Online]. Available: https://ayltoninacio.com.br/blog/criando-e-alimentando-um-cubo-olap-fisico-no-mysql. [Acedido em 9 9 2021].

[10] X. H. J. a. G. H. Li, "High-dimensional OLAP: a minimal cubing approach," em *International Conference on Very Large Data Bases*, 2004.

[11] A. Ferro, R. Giugno, P. L. Puglisi e A. Pulvirenti, "BitCube: A Bottom-Up Cubing Engineering," em *International Conference on Data Warehousing and Knowledge Discovery,* Linz, 2009.

[12] F. Leng, Y. Bao, D. Wang e Y. Liu, “An Efficient Indexing Technique for Computing High Dimensional Data Cubes,” em *International Conference on Web-Age Information Management*, Rome, 2006.

[13] R. R. Silva, J. d. C. Lima e C. M. Hirata, “qCube : efficient integration of range query operators over a high dimension data cube.,” *Journal of Information and Data Management,* vol. 4, nº 3, pp. 469-482, 2013.

[14] R. R. Silva, C. M. Hirata e L. J. d. Castro , “Big high-dimension data cube designs for hybrid memory systems,” *Knowledge and Information Systems,* vol. 62, nº 12, p. 4717–4746, 2020.

[15] R. R. Silva, C. M. Hirata e J. d. C. Lima, “A Hybrid Memory Data Cube Approach for High Dimension Relations,” em *Proceedings of the 17th International Conference on Enterprise Information Systems*, Barcelona, 2015.

[16] S. Tatham. [Online]. Available: https://www.putty.org. [Acedido em 9 9 2021].

[17] J. Blackard, D. Dean e C. Anderson, “Forest Covertype Data Set,” [Online]. Available: https://archive.ics.uci.edu/ml/datasets/covertype. [Acedido em 15 6 2021].

[18] J. Tromp, “Connect-4 Data Set,” [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Connect-4. [Acedido em 15 6 2021].

[19] “Exame Data Set,” [Online]. Available: repositoriodatasharingfapesp.uspdigital.usp.br/handle/item/2. [Acedido em 15 6 2021].

[20] [Online]. Available: https://dl.acm.org/doi/10.5555/1316689.1316736. [Acedido em 9 9 20201].

ANEXOS

Anexo A: PROPOSTA DE ESTÁGIO

Ano Letivo de 2020/2021
2º Semestre

Redução de IDs em Cubos de Dados

SUMÁRIO

Computar um cubo de dados é uma tarefa essencial, dado que a pré-computação de parte ou de todo o cubo de dados pode reduzir substancialmente o tempo de execução e melhorar o desempenho de sistemas OLAP.

Alguns trabalhos empregam as técnicas de índice invertido o trabalho mais significativo que implementa este conceito é o Frag-Cubing (LI; HAN; GONZALEZ, 2004) onde cada tupla *iT* tem um valor de atributo, uma lista de identificadores da tupla (TIDs) e um conjunto de valores de medida. Assim este projeto tem o objetivo explorar como as listas de TIDs podem ser reduzidas, pois valores de atributos muito frequentes sequencialmente pode gerar problemas de espaço de armazenamento em memória externa quando esta for limitada a poucos gigabytes.

Este projecto destina-se a um(a) aluno(a).

RAMO: Indicar o(s) ramo(s) em que se enquadra:

☒ Desenvolvimento de Aplicações
☐ Redes e Administração de Sistemas
☒ Sistemas de Informação.

1. ÂMBITO

Sistemas OLAP dependem em sua maioria da computação de um cubo de dados, uma vez que a pré-computação de parte ou de todo o cubo de dados pode reduzir substancialmente o tempo de execução e melhorar o desempenho destes sistemas. Porém, esta tarefa é um dos problemas mais importantes e pesquisados na área de DW. Já que o problema possui complexidade exponencial em relação ao número de dimensões, a materialização completa de um cubo envolve uma grande quantidade de células e um elevado tempo para sua geração.

Computar um cubo de dados é uma tarefa essencial, dado que a pré-computação de parte ou de todo o cubo de dados pode reduzir substancialmente o tempo de execução e melhorar o desempenho de sistemas OLAP. Porém, esta tarefa é um dos problemas mais importantes e pesquisados na área de DW. Já que o problema possui complexidade exponencial em relação ao número de dimensões, a materialização completa de um cubo envolve uma grande quantidade de células e um elevado tempo para sua geração. Assim, é um problema manter um cubo de dados atualizado segundo algum intervalo de tempo. Uma abordagem para cubo de dados não é completamente prática sem as garantindo que células sejam organizadas hierarquicamente e garantir consultas diversas a tais células de um cubo.

Alguns trabalhos empregam as técnicas de índice invertido e índice bitmap para computar cubos com alta-dimensionalidade (LI; HAN; GONZALEZ, 2004; LENG; et

al., 2006; WU; STOCKINGER; SHOSHANI, 2008; FERRO; et al., 2009; LO; et al., 2008; SILVA; LIMA; HIRATA, 2013; SILVA; LIMA; HIRATA, 2015a).

Frag-Cubing (LI; HAN; GONZALEZ, 2004) implementa o conceito de índice invertido onde cada tupla *iT* tem um valor de atributo, uma lista de identificadores da tupla (TIDs) e um conjunto de valores de medida. Por exemplo, consideremos quatro tuplas: $t_1 = (tid_1, a_1, b_2, c_2, m_1)$, $t_2 = (tid_2, a_1, b_3, c_3, m_2)$, $t_3 = (tid_3, a_1, b_4, c_4, m_3)$, e $t_4 = (tid_4, a_1, b_4, c_1, m_4)$. Estas quatro tuplas geram oito tuplas invertidas: iTa_1, iTb_2, iTb_3, iTb_4, iTc_1, iTc_2, iTc_3 e iTc_4. Para cada valor de atributo é construído uma lista de ocorrências, assim para a_1 temos $iTa_1 = (a_1, tid_1, tid_2, tid_3, tid_4, m_1, m_2, m_3, m_4)$ onde o valor de atributo a_1 está associado aos TIDs: tid_1, tid_2, tid_3 e tid_4. O identificador de tupla tid_1 tem o valor de medida m_1, tid_2 tem o valor de medida m_2, tid_3 tem o valor de medida m_3 e tid_4 possui o valor de medida m_4. A consulta $q = (a_1, b_4, COUNT)$ pode ser respondida por $iTa_1 \cap iTb_4 = (a_1b_4, tid_3, tid_4, COUNT\ (m_3, m_4))$. Em q, $iTa_1 \cap iTb_4$ indica os TIDs comuns em iTa_1 e iTb_4.

Este projeto tem o objetivo explorar como as listas de TIDs podem ser reduzidas, pois valores de atributos muito frequentes sequencialmente pode gerar problemas de espaço de armazenamento em memória externa quando esta for limitada a poucos gigabytes. Listas de TIDs muito grandes podem também onerar a consulta, uma vez que tais listas precisam ser carregadas para memória principal para depois realizar operações de interseção. Com isto, explorar diferentes representações para listas de TIDs é algo relevante para inúmeras abordagens de cubos de dados baseadas em índice invertido.

2. OBJECTIVOS

Propor uma estrutura de para redução da representação de índices. Uma possível solução seria representar intervalos de TIDs para valores de atributos contínuos, ou seja, que aparecem em tuplas consecutivas na relação. Normalmente, relações com dimensões com baixa cardinalidade e skew alto podem conter valores de atributos contínuos. Dada uma dimensão com o atributo SEXO em uma relação de dados que armazena registros de um salão de beleza. Nesta relação a frequência do valor de atributo F é alta e representa grandes intervalos de tuplas, assim uma relação como a apresentada pela Tabela 1 pode ser reduzida conforme ilustra Tabela 2.

TABELA 1 – Relação de clientes

TID	Nome	Sexo
1	CARLA	F
2	JOANA	F
3	RUTH	F
4	NADIR	F
5	LAIS	F
6	JOÃO	M
7	DALVA	F
8	DOLORES	F

TABELA 2 – Relação com dimensão sexo representada por intervalo de TIDs

Dimensão	Valor de Atributo	BID	TID
SEXO	F	1	1..5
	M	2	6
	F	2	7..8

Um problema desta estratégia são as atualizações. Na alternativa de solução proposta a ideia para que o intervalo não tenha que ser reconstruído seria manter a Tabela 1 inalterada e marcar a tupla alterada em uma tabela auxiliar, como ilustra a Tabela 3, onde o valor de atributo da dimensão SEXO que foi alterado para a tupla com *tid*$_4$ é mantido. Estas e outras estratégias para redução de TIDs devem ser investigadas e incorporadas na abordagem bCubing, tornando-a cada vez mais robusta e útil para problemas OLAP reais.

TABELA 3 – Relação com valores de atributos alterados indexados pelo TID

Dimensão	BID	TID	Valor de Atributo
SEXO	1	4	M

…

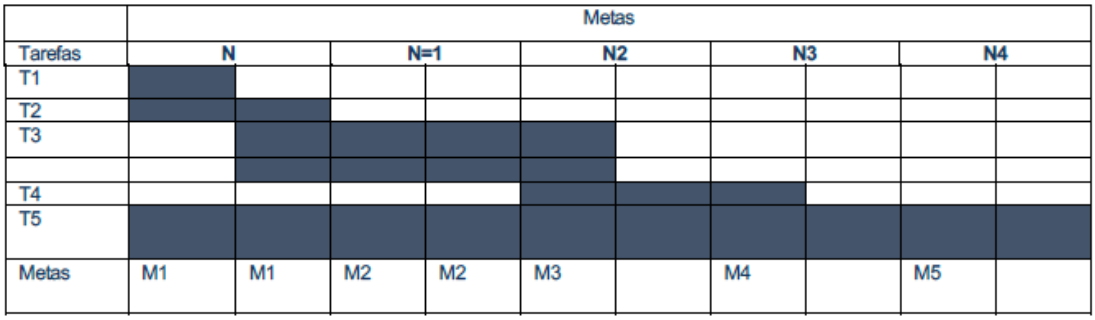
3. PROGRAMA DE TRABALHOS

O projecto/estágio consistirá nas seguintes actividades e respectivas tarefas:

- ***T1*** – *Pesquisa bibliográfica* – OLAP e cubo de dados;
- ***T2*** – *Testes com Abordagens Clássicas* – Neste ponto pretende-se que seja construído um ambiente de testes com abordagems classicas de computação de cubos de dados.
- ***T3*** – *Desenvolvimento* – Desenvolvimento de uma abordagem para redução de índices em abordagens de índices invertidos;
- ***T4*** — *Escrita de position paper* – Escrita e submissão de position paper com problema e abordagem a adotada;
- ***T5*** – *Análise dos resultados* – Análise dos resultados obtidos. Produção de tabelas, gráficos e grelhas comparativas.
- ***T6 –*** *Elaboração do Relatório e do Artigo* – Preparação do relatório final. Submissão de um artigo científico a uma conferência internacional. A elaboração do relatório e do artigo deverão ser efectuadas gradualmente ao longo do projeto.

4. CALENDARIZAÇÃO DAS TAREFAS

O plano de escalonamento dos trabalhos é apresentado em seguida:



INI Início dos trabalhos
M1 (INI + 3 Semanas) Tarefa T1 terminada

M2 (INI + 6 Semanas) Tarefa T2 terminada

M3 (INI + 22 Semanas) Tarefa T3 e T4 terminada
M4 (INI + 18 Semanas) Tarefa T5 terminada
M5 (INI + 24 Semanas) Tarefa T6 terminada

5. LOCAL E HORÁRIO DE TRABALHO

O trabalho decorrerá no Centro de Informática e Sistemas da Universidade de Coimbra (CISUC) ou, devido às condicionantes impostas pelo COVID-19, em teletrabalho.

6. TECNOLOGIAS ENVOLVIDAS

O aluno poderá optar por uitlizar uma das seguintes linguagens: C, C++, Java, C# ou Pynthon

7. METODOLOGIA

Será organizado um Dossier de Projeto, no qual será documentado todo o progresso do projecto e que servirá de entrada para o Relatório de Projeto. Reuniões de coordenação semanais: Discussão e apresentação de resultados, brainstorming, avaliação e coordenação dos trabalhos.

8. ORIENTAÇÃO

CISUC e FATEC-MC:
Rodrigo Rocha Silva (rrochas@dei.uc.pt)
Professor e Investigador
ISEC:
Jorge Bernardino (jorge@isec.pt)
Professor Coordenador

Anexo B: Artigo científico

CiiCube: A Compressed Inverted Index Approach to Data Cubes

Abstract

The increase in the amounts of information used to do data analysis is problematic due to the fact that a single system may not have the necessary memory to run the analysis. In this paper we present a Compressed Inverted Index Data Cube (CiiCube), that can index and query data cubes with a high number of tuples and dimensions in a single machine. CiiCube was evaluated in considering runtime and memory consumption. The tests executed showed that CiiCube's compression algorithm works better the higher the data set's skewness. Notably, when the data set skew is equal to 5, CiiCube's was able to index a data set with 500 million tuples using less than 12 gigabytes of memory.

1. Introduction

In the last decades, the information systems popularity increased exponentially, increasing the amount of data that is collected. Services that used to be done in person are now done online, companies now attempt to give fully personalized services by analysing their costumers' data patterns and also scientific research is done by analysing massive amounts of data. In order to be able to do the analysis, companies need ever increasing computing power, since the amount of data available has been increasing for the last decades [1, 2].

The rise in the size of such data collections has been outgrowing the increase in processing power that a single system can process, resulting in the need to have multiple computers to do a single analysis [3].

Due to the reasons explained above, there is a clear need for algorithms that use less memory and can process data faster, which is something quite hard to get, since, as a general rule of thumb, in computer science, there is almost always a trade-off between speed and size [4]. The ability that some algorithms have to "*consolidate, view and analyse data according to multiple dimensions, in ways that make sense to one or more specific enterprise analysts*" [5] is called OLAP (online analytical processing).

In order to be perform such activities, the algorithms need to create a multiple-dimensional representation of the data, usually utilizing arrays, known as data cube, and then perform the analysis.

Since before being able to do any kind of analyses it is needed to create the data cube structure, that can utilize gigabytes of memory by itself, we consider that the main concern to new algorithms should be to reduce the memory needed to create and process data cubes.

In 2006, Frag-Cubing [6] was presented as a viable option to perform OLAP. Frag-Cubing was the first OLAP algorithm with a good runtime using an acceptable amount of memory to index the data sets, it used inverted index tables to create the data cube and used iceberg cubing computation, presented in [7], to compute queries.

Many algorithms are based on the inverted index approach presented in Frag-Cubing, introducing new operations to what Frag-Cubing already could do, such as HFRAG [8], or changing its structure and type of memory usage in order to improve memory consumption, such as bCubing [13].

The algorithm presented in this paper, CiiCube, is also based on Frag-Cubing. It uses the basic notion that is not always necessary to represent all the different values of a range, instead, in cases such as a continuous values range, it is possible to represent the range simply by using its lower and higher values and the increment between its numbers.

With this work, we expect to show how the compression of inverted index lists affects both runtimes and memory usage, that can be useful to further work being made, both in the areas of research and real-world applications.

Experiments done with real and synthetic data shown that high skewed data allows CiiCube to use considerably less memory than Frag-Cubing. Our work also shows that, in a worst-case scenario to runtime, CiiCube is around 4 times slower to answer queries when the dimensional skew is 2. When the data skew increases to 5, however, CiiCube uses around 4 times less memory to create the data cube, while being almost as fast as Frag-Cubing to answer the queries done.

The rest of the paper is organized as follows: Section 2 details related work, such as other OLAP algorithms based on inverted index and bitmap. Section 3 details CiiCube approach, detailing its structure and most relevant algorithms. Section 4 describes the experiments done comparing CiiCube and Frag-Cubing and discusses the results. In Section 5 we conclude our work and point both to possible improvements and observations.

2. RELATED WORK

Due to the increasing amount of data being tracked and stored, OLAP analysis is an increasing problem created by nowadays technology. This problem can be divided in two subproblems. The first one is the creation of the data cube structure itself, that can use tens or even hundreds of gigabytes of RAM memory, this first sub-problem is the data cube indexation. The second subproblem is to answer any queries, since to be able to answer queries both extra memory and processing power are required.

In order to solve this problem, there are two main approaches that implement a sequential high dimension cube solution: the usage of a bitmap-based structure such as [10, 5, 9] and the use of inverted index-based structure [6, 8, 11].

In this section we will expose multiple different algorithms designed with the objective of allowing OLAP analysis. The algorithms here presented are bitmap-based, inverted index-based and binary three-based.

BitCube [10] is an algorithm that uses bitmaps to store and identify tuple attributes in a data cube. BitCube sections a data cube by its dimensions and then sections its dimensions into attribute values. For each attribute value a bitmap is created, this is, for each attribute value, an array of binary values (bitmap) with the number of tuples is created where each value represents a tuple. In the bitmap, if the tuple has the value represented by that bitmap, it stores the bit ‘1’, otherwise, stores ‘0’. This approach is quite effective for data cubes with a low or moderate number of tuples. When using bigger data sets, however, the processing time and memory required can get quite high, due to having both memory and processing runtime increasing exponentially with the growth of the number of dimensions and cardinality, as it was shown in the paper.

Compressed Bitmap Index Based Method [12] is an algorithm were, as the name suggests, a data cube is represented using bitmap arrays with compression. In this compression, two different pointers are used to delimit the first and last position where the bit one appears in the bitmap, only representing the bitmap values whose positions are inside the interval delimited by those to pointers. The tests done in the paper have shown this algorithm is faster and uses less memory than Frag-Cubing to process and store data sets with a high number of dimensions and low cardinality. The usual bitmap limitations with high cardinality data sets are still present, whereas inverted index-based algorithms, such as Frag-Cubing, does not suffer from that problem.

Frag-Cubing [6] is an approach that uses inverted index tables to store and process the data cube. Each tuple is composed by its id (TID) and a set of attribute measures. For each attribute measure, a list with all the attribute values is created. Each attribute value

is related with a TIDs' list, being in that list the TIDs of the tuples that have such attribute value. This work has been further improved in algorithms such as H-FRAG, qCube, bCubing. As it was shown, Frag-Cubing works very well in data sets with 10^6 tuples, however, in the environment of Big Data, single systems struggle to be able to compute data sets with 10^7 or more tuples using Frag-Cubing. This algorithm's memory consumption grows linearly along the increase in the number of dimensions and tuples [6, 11]. Another problem is to process queries, since, as an example, the memory needed to answer subcube queries is not negligible.

qCube, [8], also uses inverted indexes to compute range queries over high dimension data cubes. Its design, just like Frag-Cubing, uses sorted intersections and unions to do OLAP computing and has a linear memory and runtime as the number of attributes per tuples (dimensions) increase. It also implements multiple operators, such as point, range, and inquire queries.

H-FRAG [11] utilizes a hybrid memory system, distributing the tuples and TID lists smartly between main memory and disk. When the data cube is first being created H-FRAG starts by deciding which fragments of the cube are stored in main memory and which ones are stored in disk. To do that, H-FRAG scans the entire data set to obtain the frequency of each attribute value, for each dimension in the data set. After that, the average frequency is calculated, attributes with a frequency above the average are stored in the system's main memory while the others are stored in external memory. Another difference between frag-cubing and H-FRAG is the fact that while frag-cubing only implements equal and sub-cube query operators, H-FRAG also implements range queries. The biggest problem in H-FRAG is the poor runtime performance when processing small relations, comparing with main memory-based algorithms, such as Frag-Cubing.

bCubing [12] utilizes both main memory and disk to store and process data, such as H-FRAG [11], however, the management of the hybrid data is completely different. The main difference between bCubing and most other Frag-Cubing based algorithms is its structure. While most algorithms have a single array structure used to store and access the data cube, bCubing uses two different array structures: one to store the data cube itself and a second used to map the first structure, so the access is more efficient. The tuples are divided into blocks, each block is identified by its id, or, for short, BID. The blocks have a maximum number of tuples, and, except a single last block, all blocks have that size. A first table, kept in the disk, is used to store the tuple ids and attribute values is also created, dividing them by the blocks. A second table, which is stored in main memory, is used to map the attribute values of each block, allowing the program to know if the block contains an attribute before accessing it. The experiments done show that, when compared to the Frag-Cubing approach, bCubing becomes more efficient than Frag-Cubing to

process bigger data cubes, even allowing to process and store data sets with 10^9 tuples. It also must be pointed out that this algorithm still suffers from the same high performance penalty when answering smaller relations

The work done in [14] is quite different from all the algorithms explained above. In that paper, the authors create a data cube using a binary search three, named Binary Search Prefix Three (BSPT), to store the cuboids. To support the BSPT table, the definitions of "prefix" and suffix are created. A prefix is a table value that comes before the value being analysed, a suffix is a value that comes after the value being analysed. The structure used to do the BSPT tree is composed by:

1. The attribute value being represented in that object;
2. Two child attribute values, that can only store other attribute values of the same dimension;
3. A child attribute value that stores an attribute value of a the next dimension;
4. A list to store the tuple identifiers that contain the attribute values being represented until that part of the tree.

Must be noted that only different dimensions' attribute values' combinations (point 3) are stored in the BSPT three. This algorithm stores the BSPT table in disk. In order to answer any queries, the author proposes an algorithm similar to the ones used on binary threes. Unfortunately, the experiments done in the paper only considered this algorithm, therefore we cannot conclude its competence and capabilities when compared with other algorithms.

Further improvements over the algorithm presented in [14] have been done by the same author in [15], [16], [17] and [18].

The CiiCube algorithm being presented in this paper is based on Frag-Cubing. Therefore, utilizes only the computers' main memory to store the inverted indexes. Because the changes are done at level of the inverted index lists, our work is compatible with all the inverted index-based algorithms presented above.

3. THE CIICUBE APPROACH

In this section, we present a new structure named CiiCube, that uses the system's main memory to store, update and query Big Data cubes, employing compression to reduce the amount of memory necessary to represent a data cube, therefore enabling systems to process bigger data cubes than what they could using regular inverted index tables.

3.1. CiiCube Representation

Just like Frag-Cubing, CiiCube's structure divides the data cube into dimensions, and, for each dimension, creates lists. Each list is related to an attribute value and is used to store the ids of tuples (TIDs) that contain the attribute value. All the inverted index-based algorithms we are aware of, such as Frag-Cubing, H-FRAG, qCube and bCubing, utilize a single array as the list to store the TIDs.

Our proposal is related to how the TIDs are stored in the inverted index lists. We recon it is not necessary to represent all the values of a list, for example, when some of them are numerically followed, it is possible to only represent the interval by only storing the lower and higher TIDs. Furthermore, it is not always possible to compress the TIDs, in that case we decided to store those TIDs in a regular way.

In our proposal, instead of storing the TIDs into a single array, CiiCube uses the class *DIntArray*. The class *DIntArray* consists of three different arrays, named as *noReductionArray, reductedPos1* and *reductedPos2*.

noReductionArray is used to store the TIDs that cannot be compressed. Note that we only compress groups of three or more TIDs, therefore, two TIDs with followed values do not create an interval. The other two arrays created are used to represent the compressed TIDs. The array *reductedPos1* stores the lower TID of the interval and the array *reductedPos2* stores the higher TID of the same interval. The intervals are connected by being in the same position in both arrays, consequently, both arrays have always the same size.

As an example, let us imagine that the list of TIDs *(1, 2, 4, 7, 8, 9, 10, 13, 15, 16, 17, 18, 19)* have some attribute value in common. The usual single array approach would create a single array with those values. The proposed *DIntArray* structure, however, how have got the following composition:

- *noReductionArray - [1, 2, 4, 13];*
- reductedPos1 - *[7, 15];*
- *reductedPos2 - [10, 19]*.

3.2. CiiCube's Data Cube Indexation

Programs such as Frag-Cubing, usually, use single arrays to store the inverted tuples. In order to add a new TID, the new TID is directly added to the last position of its attribute value's TID list. The CiiCube algorithm, however, needs to know if the TID is going to be compressed or not, therefore some extra steps need to happen. When a new TID is sent to be stored on some instance of the class *DIntArray*, three different cases can happen:

1. The TID is added to the compression;
2. A new compression is done;
3. The TID is added without compression.

When adding any new TID, the algorithm starts by checking if the new TID is following to the last TID stored in the array *reductedPos2* (*Figure 1*, line 1), note that this step is ignored if no TIDs have been compressed before, if it is, then the last position in the array *reductedPos2* is overwritten by the new TID. When that is not the case, the algorithm checks if it is possible to create a new compression. To do a new compression, the algorithm checks if the new TID being added and the last two TIDs stored in the *noReductionArray* array are back-to-back numbers (*Figure 1*, line 3). In the case were the new TID and the last two TIDs stored are in sequence, then the last two TID stored in *noReductionArray* are removed (*Figure 1*, line 7) and an interval composed by the former penultimate TID stored in the array *noReductionArray* and the new TID being added (*Figure 1*, lines 4 and 5). Case none of the previous conditions applies, then, the new TID is simply added to the array *noReductionArray* (*Figure 1*, lines 9 and 10).

The algorithm's pseudocode can be seen in *Figure 1*.

Input: newTid to be stored; sizeReduced which is a variable that points to the next unused position of the arrays reducedPos1 and reducedPos2; sizeNonReduced which is a variable that points to the next unused position of the array noReductionArray; reducedPos1 which stores the firt element of an interval; reducedPos2 which stores the last element of an interval; noReductionArray which stores non compressed tids;
Output: none

```
1.  if sizeReduced > 0 and  reducedPos2[sizeReduced - 1] + 1 equal newTid then
2.      reducedPos2[sizeReduced - 1] = newTid;
3.  else if sizeNonReduced > 2 and noReductionArray[sizeNonReduced - 1] + 1 equalthen  id and noReductionArray[sizeNonReduced - 2] + 2 equal newTid then
4.      reducedPos1[sizeReduced] = noReductionArray[sizeNonReduced - 2];
5.      reducedPos2[sizeReduced] = newTid;
6.      sizeReduced = sizeReduced + 1;
7.      sizeNonReduced = sizeNonReduced - 2;
8.  else
9.      noReductionArray[sizeNonReduced] = newTid;
10.     sizeNonReduced = sizeNonReduced + 1;
11. end
```

Figure 1 Indexation algorithm

The *DIntArray* structure is composed by Java arrays, which cannot have their size changed after creation. In this algorithm we decided to abstract the management of those arrays both for the sake of simplicity and the fact that is not a fundamental part of the algorithm. Having stated that, when there was the need, we decided to increase the arrays size by a factor of two. After all the tuples from the data set being added to the structure, the algorithm removed all the unused space in the arrays.

3.3. Intersection Algorithm

When answering a query, the CiiCube algorithm utilizes intersections to obtain the list of TIDs. In generic terms, the resulting TID list when looking for the tuples that contain two different attributes is obtained by using the mathematical intersection between the TID lists related to the attributes. Therefore, the intersection algorithm is one of the program's most fundamental parts and is all OLAP operations' core. An intersection algorithm with high runtime profoundly impacts the entire program and any small performance improvements can be significant when doing high runtime operations.

The CiiCube intersection algorithm can be divided into two phases. In the first phase, for both *DIntArray* objects, the lowest TID value or TID interval to be compared is defined, this is done by, in the beginning, comparing the lower TIDs to compare in the arrays noReductionArray and reductedPos1.

In the second phase the comparison between the previously chosen TIDs from different DIntArray objects is done. At this point, the comparison can be made between two different TID values, a TID value and a TID interval or two TID intervals.

In the beginning of the intersection algorithm, all the TID values and TID intervals from both *DIntArray* classes being intersected are deemed as valid to be compared. The intersection algorithm ends when one or both of the *DIntArray* classes being intersected have no more TID values or TID intervals considered valid for comparison.

If the comparison is done between two different TID values and both TIDs are equal, the TID is added to the resulting *DIntArray* object responsible to store the result of the intersection and both TID values are considered invalid for further comparisons, otherwise, the only the lower TID value is deemed invalid for further comparisons. In the case where the comparison is done between a TID value and a TID interval, it is checked if the TID value is located inside the TID interval, if so, then, the TID value is added into a *DIntArray* object responsible for storing the result of the intersection being made and is considered as invalid for further comparisons, otherwise, a comparison between both the TID value and the higher TID of the TID interval, which is stored in reductedPos2, is done and the lower of those two is considered as invalid to be used in the next comparison.

In the case where the comparison is done by two TID intervals, if one of the lower TIDs of both intervals is between the lower and higher TIDs of the other TID interval that value is going to added. The next step is to determine if it is going to be added a single TID value or a TID interval. If the TID to be added is equal to the higher TID of the other TID interval, then that lower TID is added as a single TID value without any compression, otherwise, it is added as an TID interval, the initial TID found is used as the lower TID of the new interval and the higher value of the interval to be added is the lower TID between the higher TIDs of both TID intervals being compared.

Figure 2 shows the pseudocode of CiiCube's intersection algorithm.

```
Input: DIntArray DA and DIntArray DB;
Output: DIntArray DC;

1.  aNonreduced=0;
2.  aReduced=0;
3.  bNonreduced=0;
4.  bReduced=0;
5.  While DA or DB have TIDs or TID intervals to intersect do
6.     If DA.sizeNonReduced equal aNonreduced or DA.reducedPos1[aReduced] < DA.sizeNonReduced[aNonReduced] then
7.        if DB.noReductionArray equal bNonreduced or DB.reducedPos1[bReduced] < DB.noReductionArray[bNonReduced] then
8.           if DA.reducedPos1[aReduced] >= DB.reducedPos1[bReduced] and DA.reducedPos1[aReduced] <= DB.reducedPos2[bReduced] then
9.              If DA.reducedPos1[aReduced] equal DB.reducedPos2[bReduced] then
10.                adds the value DA.reducedPos1[aReduced] to DC and increments aReducted and bReducted;
11.             else
12.                if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
13.                   adds the interval [DA.reducedPos1[aReduced]; DA.reducedPos2[aReduced]] to DC and increments aReduced;
14.                else
15.                   adds the interval [DA.reducedPos1[aReduced]; DB.reducedPos2[bReduced]] to DC and increments bReduced;
16.                end
17.             end
18.          else if DB.reducedPos1[aReduced] >= DA.reducedPos1[bReduced] and DB.reducedPos1[aReduced] <= DA.reducedPos2[bReduced] then
19.             If DB.reducedPos1[bReduced] equal DA.reducedPos2[aReduced] then
20.                adds the value DB.reducedPos1[bReduced] to DC and increments aReducted and bReducted;
21.             else
22.                if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
23.                   adds the interval [DB.reducedPos1[bReduced]; DA.reducedPos2[aReduced]] to DC and increments aReduced;
24.                else
25.                   adds the interval [DB.reducedPos1[bReduced]; DB.reducedPos2[bReduced]] to DC and increments bReduced;
26.                end
27.             end
28.       else if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
29.          increments aReduced;
30.       else
31.          increments bReduced;
32.       end
33.    else if DB.reducedPos1 equal bReduced or DB.reducedPos1[bReduced] > DB.noReductionArray[bNonReduced] then
34.       if DB.noReductionArray[bNonReduced] >= DA.reducedPos1[aReduced] and DB.noReductionArray[bNonReduced] <= DA.reducedPos1[aReduced] then
35.          adds the value DB.noReductionArray[bNonReduced] to DC and increments the counter bNonReduced;
36.       else if DB.noReductionArray[bNonReduced] <= DA.reducedPos2[aReduced] then
37.          increments bNonReduced;
38.       else
39.          increments aReduced;
40.       end
41.    end
42. else if DA.reducedPos1 equal aReduced or DA.reducedPos1[aReduced] > DA.noReductionArray[aNonReduced] then
43.    if DB.noReductionArray equal bNonreduced or DB.reducedPos1[bReduced] < DB.noReductionArray[bNonReduced] then
44.       if DA.noReductionArray[aReduced] >= DB.reducedPos1[bReduced] and DA.noReductionArray[aReduced] <= DB.reducedPos1[bReduced] then
45.          adds the value DA.noReductionArray[aNonReduced] to DC and increments aNonReduced;
46.       else if DA.noReductionArray[aNonReduced] <= DB.reducedPos2[bReduced] then
47.          increments aNonReduced;
48.       else
49.          increments bReduced;
50.       end
51.    else if DB.reducedPos1 equal bReduced or DB.reducedPos1[bReduced] > DB.noReductionArray[bNonReduced] then
52.       if DB.noReductionArray[bNonReduced] equal DA.noReductionArray[aNonReduced] then
53.          adds the value DA.noReductionArray[aNonReduced] to DC and increments bNonReduced and aNonReduced;
54.       else DB.noReductionArray[bNonReduced] < DA.noReductionArray[aNonReduced] then
55.          increments bNonReduced;
56.       else
57.          increments aNonReduced;
58.       end
59.    end
60. end
61. end
```

Figure 2 CiiCube's Intersection Algorithm

To ease the understanding of such algorithm, let us see, as an example, the intersection of two *DIntArray* instances:

DIntArray A, composed by:

- *noReductionArray = [2, 3, 9];*
- *reductedPos1 = [5, 12];*
- *reductedPos2 = [7, 17].*

and *DIntArray* B, composed by:

- *noReductionArray = [2, 12];*
- *reductedPos1 = [4];*
- *reductedPos2 = [10].*

The result of this intersection is going to be stored in an object named *DIntArray C*.

In order to do the intersection between these two *DIntArray* instances, the algorithm starts by determining the lowest value marked as valid to comparison from each one of them. Once again, must be noted that, in the beginning, all the TIDs are marked as valid for comparison. In the case of the *DIntArray* A, the algorithm compares 2 and 5, the lower TIDs to use from *noReductionArray* and *reductedPos1*, respectively, choosing the TID value 2 since it is lower (*Figure 2*, line 42). The same process happens in the *DIntArray* B object, where the program compares the TIDs *2* and *4*, choosing the TID value *2* (*Figure 2*, line 51). Then the algorithm proceeds to compare the chosen TIDs from the different *DIntArray* instances (*Figure 2*, line 52), the TIDs to compare are *2* and *2*, respectively from *DIntArray* A and *DIntArray* B.

Since both are TIDs and are equal, a third *DIntArray* C, responsible for storing the result of the intersection, gets to store the value *2* and both values from noReductionArray of *DIntArray* A and B are marked as invalid for further comparison (*Figure 2*, line 53).

The algorithm needs to, again, choose which TIDs are going to be compared next. In the case of *DIntArray* A, the TIDs being compared are *3* and *5*, respectively, from *noReductionArray* and *reductedPos1*, choosing *3* for being lower (*Figure 2*, line 42). In the case of *DIntArray* B, the TIDs being compared are *12* and *4*, respectively, from *noReductionArray* and *reductedPos1*, choosing *4* for being lower (*Figure 2*, line 43). Then the algorithm needs to do the intersection between the values chosen previously.

Note, that, in this case, the comparison being made, is between a TID value, *3*, from *DIntArray* A, and a TID interval, *[4 ; 9]*, from *DIntArray* B. Since *3* does not belong to the interval *[4; 9]* and 3 is lower than 4, the TID *3* is marked as invalid for further comparisons (*Figure 2*, lines 46 and 47).

Right after that, the algorithm then chooses again which number is to be compared next. In the case of *DIntArray* A, the TIDs to be compared are *9* and *5*, respectively, from *noReductionArray* and *reductedPos1*. Since *5* is lower than *9*, the TID interval *[5; 7]* is the one being compared (*Figure 2*, line 6). In the case of *DIntArray* B, the comparison done is the same as the one done before and its result is the same, therefore the TID interval *[4; 10]* will be used for comparison again (*Figure 2*, line 7).

In the following step, the algorithm does the comparison of the TID intervals of different arrays. The TID intervals being compared are *[5;7]* and *[4;10]* from, respectively, *DIntArray* A and B. In this case, the TID *5*, lowest TID from one of the intervals, is in between the values of the TID interval *[4; 10]* (*Figure 2*, line 8), besides that, the value *5* is lower than the value *10* (*Figure 2*, line 9), which is the highest value of the interval *[4; 10]*, so the algorithm knows it needs to add an TID interval with the value *5* as the lower value and *7* as the highest value, since *7* is lower than *10*, therefore, the TID interval *[5, 7]* is added to the *DIntArray* C (*Figure 2*, lines 12 and 13). Note that the TID interval *[4; 10]* is going to be reused in further comparisons, since the higher TID of this interval is higher than the one in the interval *[5; 7]* (*Figure 2*, line 13).

Next, the algorithm, once again, needs to choose the TIDs of *DIntArray* A and B that are going to be intersected. In the case of the *DIntArray* A, the option is between the TIDs *9* and *12*, from, respectively, *noReductionArray* and *reductedPos1*, choosing the TID value *9* (*Figure 2*, line 42). In the case of *DIntArray* B, the same TID interval *[4; 10]* gets to be used again (*Figure 2*, line 43).

Afterward the algorithm does the comparison of the TID value *9* with the TID interval *[4; 10]*, resulting in adding the TID value *9* to the resulting *DIntArray* C (*Figure 2*, lines 44 and 45). After that, the TID value 9 is marked as invalid for further comparisons., while the TID interval *[4; 10]* is kept as valid for further comparisons (also done in *Figure 2*, line 45).

Following that, the algorithm does not have any TIDs left to choose from the array *noReductionArray* of the *DIntArray* A, therefore is forced to use the lower valid TID interval, which is *[12; 17]* (*Figure 2*, line 6). The *DIntArray* B will, again, use the TID interval *[4; 10]* (*Figure 2*, line 7). As it is noticeable, the result of the intersection between the TID intervals *[12; 17]* and *[4; 10]* is null, therefore the program simply decides which of the TID Intervals is going to be marked as invalid for further comparison, to do that, the algorithm compares the higher values of both intervals, *10* and *17*, and, since *10* is lower than *17*, decides that the interval *[4; 10]* is the one marked as invalid (*Figure 2*, lines 30 and 31).

Immediately after, the algorithm, once again, needs to obtain the TIDs to do the intersection. In the case of the *DIntArray* A, it is going to be used, again, the TID interval

[12; 17] (*Figure 2*, line 6), while, in the *DIntArray* B, the only TID value remaining is *12* (*Figure 2*, line 33).

The algorithm, then, does the comparison between the TID interval *[12; 17]* and the TID value *12*, adding in the *DIntArray* C the TID value *12* and marking that TID value as invalid for further comparisons (*Figure 2*, line 6).

At this point, there is no values or intervals to be intercepted within the *DIntArray* B, so the algorithm returns the contents of the *DIntArray* C (*Figure 2*, line 5).

The resulting *DIntArray* C is:

- *noReductionArray = [2, 9, 12];*
- *reductedPos1 = [5];*
- *reductedPos2 = [7].*

This intersection algorithm requires more computing power than the algorithm used to intersect single TID arrays, as used in the Frag-Cubing algorithm, since it needs to choose if it will use a TID interval or a TID value to intersect. However, it is possible that, in some cases, can be faster than the Frag-Cubing intersection algorithm since it can process TID intervals at once. It is also expected that this algorithm has almost the same runtime as Frag-Cubing's intersection algorithm in cases where almost no compression is done. Real-world analyses are done in the experiments section.

3.4. Update Algorithm

There are two main kinds of updates in data cubes: adding a new tuple or modifying the attribute value of one or more dimensions of an already added tuple. Introducing a new tuple into a data cube uses the same process as normal indexation, which has already been explained in section 3.2.

Modifying an already existing tuple without the need to reprocess the entire cube, due to the modifications that can happen to the intervals, requires some extra structure.

When dealing with *DIntArray* classes that contains millions of TIDs it is quite an expensive operation to re-process the entire *DIntArray* class so that a single tuple can be removed or added. Instead of doing the re-process, for each dimension, extra two arrays are created, one to store the TIDs with the modified value and another to store the modified value itself, *tidModified* and *valueModified*, respectively. Note that the TIDs stored in the array *tidModified* are stored and added in sequence, from the lower TID to the higher, easing the retrieval of the values.

When modifying the attribute values of a tuple, the program receives the TID to be modified and all the new attribute values for the dimensions. The algorithm, then, compares, for each dimension, the new attribute value with the original attribute value of

the data cube. If both attribute values are the same, the algorithm knows that TID is not modified, however, that TID could have been modified before in that dimension, so, it looks into the *tidModified* array and if it finds the TID being modified there, removes it and its value from the arrays *tidModified* and *valueModified*, respectively.

In the case where, for some dimension, the new attribute value is different from the one existing in the original data cube, the program searches for the TID into the array *tidModified*. This search is done in the case were a TID is being re-modified. If the TID being modified is found in the array *tidModified*, the algorithm simply updates the correspondent modified value stored in the array *valueModified*. Otherwise, if the TID being modified is not found in the array *tidModified*, the algorithm adds the newly modified TID and its new attribute value into the *tidModified* and *valueModified* arrays, introducing both in a position where the TIDs being stored are in sequence.

The algorithm to modify a single tuple is shown in *Figure 3*:

Input: the *TID* being modified and its *newValue*; *tidModified*, which is an array that stores the modified tids; *valueModified*, which is an array that stores the new values of the modified tids; *sizeModified*, wich is a numerical variable used to store the number of modified tids stored.
Output: none;

```
1.  DataCubeValue = the attribute value of TID in the main data cube;
2.  if newValue equal DataCubeValue then
3.      for each T in tidModified do
4.          if T equal TID then
5.              removes T and the correspondent value in valueModified;
6.              return;
7.          end
8.      end
9.  end
10. for I from 0 to sizeModified do
11.     if tidModified[I] equal TID then
12.         valueModified[I] = newValue;
13.         return;
14.     end
15. end
16. for I from 0 to sizeModified do
17.     if tidModified[I] > TID then
18.         pushes the values with index >= I a single position to the right in the arrays tidModified and valueModified;
19.         tidModified[I] = TID;
20.         valueModified[I] = newValue;
21.         sizeModified = sizeModified + 1;
22.     end
23. end
```

Figure 3 CiiCube's update Algorithm

Must be noted that using an external structure, such as the one explained, to store modified attribute values will force the algorithm always to have extra processing when retrieving the TIDs' list of an attribute value.

When algorithm wants to retrieve the TID list of some attribute value, if the array *tidModified* is empty, then the algorithm simply returns the attribute value's related *DIntArray* object. However, when there are modified TIDs, a process to determinate

which of the TIDs should be added or removed from the *DIntArray object* to be returned is done. The reason why the TIDs are in numerical order is to allow the use of a single loop to do that operation.

All the TIDs in the array *tidModified* are checked in order to know if they should be added to the returning *DIntArray* or removed from it.

In the cases where a TID can be found both in the attribute value's returning *DIntArray* object and in the array *tidModified*, that TID is removed from the returning *DIntArray* object. Otherwise, if a TID stored in the array *tidModified* is not found in the attribute value's returning *DIntArray* object, the algorithm checks the modified attribute value related to that TID. If the modified attribute value related to the TID is equal to the attribute value whose TIDs list is being retrieved, then, the algorithm adds that TID to the *DIntArray* object being returned.

3.5. Queries Implemented

Having in account the need to test the algorithm, we decided to implement two kinds of queries: point queries and subcube queries. The query syntax uses three different operators:

- Aggregate operator: this operator is used to indicate no restrictions to the attribute value of some dimension. Its syntax is '*';
- Equal operator: this operator is used to instantiate an attribute value that must be present in the tuples to be returned. Its syntax is the instantiated attribute value itself;
- Inquire operator: this operator is used to indicate that some dimension is being inquired. Inquire operators are only present in subcube queries. Its representation is '?'.

Point queries search for the list of tuples that contain the attribute values defined by equal operators in the query. When answering this query, the algorithm obtains the TID lists related to each of the instantiated attribute values defined in the query made. Then it uses the intersection method to obtain the ones that contain all the attribute values defined. The output shown to the user is the count of the resulting TID list.

Subcube queries are operations used to analyse part of the data cube. When answering this query, the algorithm obtains the list of TIDs that contain the instantiated attribute values. Next, the algorithm creates a subcube composed by the TIDs list obtained in the step before. Finally, the algorithm seeks to answer every single combination of point query varying the values of the inquired dimensions, using the created subcube to do that.

4. Experiments

Multiple testes were done in order to compare CiiCube's approach with Frag-Cubing. Both Frag-Cubing and CiiCube were written in java (version 16) and attempt to be as similar as it was possible in order to understand how the proposed structure compares to the regular single array TID list structure both in runtime and memory consumption, minimizing other differences such as slightly different algorithm approaches.

4.1. Experimental Setup

All the testes were run on a system with an AMD Epyc processor, virtually reduced to 4 cores, 32 gigabytes of RAM.

The Java command "Xmx30g" was also used in all the tests. This command indicates that the programs can use a maximum of 30 gigabytes of memory, and it was necessary since, without it, the Java programs would not use more than 8 gigabytes of memory. The value of 30 gigabytes was used since using more than that would often make the system use swap operations and even kill the Java process due to lack of resources.

In other to create the synthetic data sets a generator provided by the IlliMine project was used. This program allowed us to change parameters such as number of tuples, number of attribute values per tuple, which will be named as number of dimensions, cardinality of each dimension and the general skew of the data set.

For the remainder of this section, **T** is the number of tuples, **D** is the number of dimensions, **C** is the cardinality of the dimensions, which, for practical reasons is equal to the biggest attribute in a dimension, and **S** is the skew of the attributes, were S=0 means that the attributes distribution is uniform along the data set and, as S becomes greater, the data gets more skewed towards a random central value.

The tests made obtained the speed to index the data cube, its size in memory after the operation being completed, the runtime of different query operations and the memory used by the algorithm to process such operations.

In order to obtain the results an industry standard procedure was used: all the tests were run five times, the lower and higher values were removed, and the result is an average of the remaining three values. The result obtained is the value that well represent the metrics explained bellow.

Finally, both algorithms have two verbose options. Verbose is an option that informs the program if the user wants the results to be shown. If verbose is on, the result of subcube queries is shown, whereas if verbose if off, then the subcube operation ends

without displaying the results. Because showing the results is quite slow and unnecessary to this study, verbose was kept off.

4.2. Indexation Runtimes and Structure Sizes

Indexation tests are used to test the differences between CiiCube and Frag-Cubing algorithms when creating the data cube. These tests are measured with two different metrics:

- Indexation Runtime: this metric is used to analyse the amount of time needed for both algorithms to create the data cube.
- Structure Size: this metric is used to analyse the amount of memory necessary to keep the finalized data cube in memory. Note that indexing a data cube can momentarily consume more memory than the value obtained, since the last operation done by the indexation algorithm is deleting the space unused by the structure.

Regarding the indexation runtimes, both algorithms had linear growth with the number of tuples, number of dimensions and cardinality. In the tests with varying skew, both programs had better indexation runtimes with higher skews. As it was expected, the CiiCube algorithm was, generally, slightly slower to index the data cube when compared to Frag-Cubing, which is expected since it needs to do more computation to do the compression.

The only variable that seems to change the pattern of faster indexation runtimes from the Frag-Cubing algorithm is skew. In *Figure* 4 it is possible to see the evolution of the indexation runtime varying the skew for both algorithms. As it is possible to see, the increase of the skew is followed by a general decrease in the gap between both programs.

The decrease in the indexation runtimes from CiiCube, when compared to the Frag-Cubing algorithm, happens due to the fact that CiiCube does less times the operation of increasing the size of the inverted index arrays, which, as explained above, is the slowest operation.

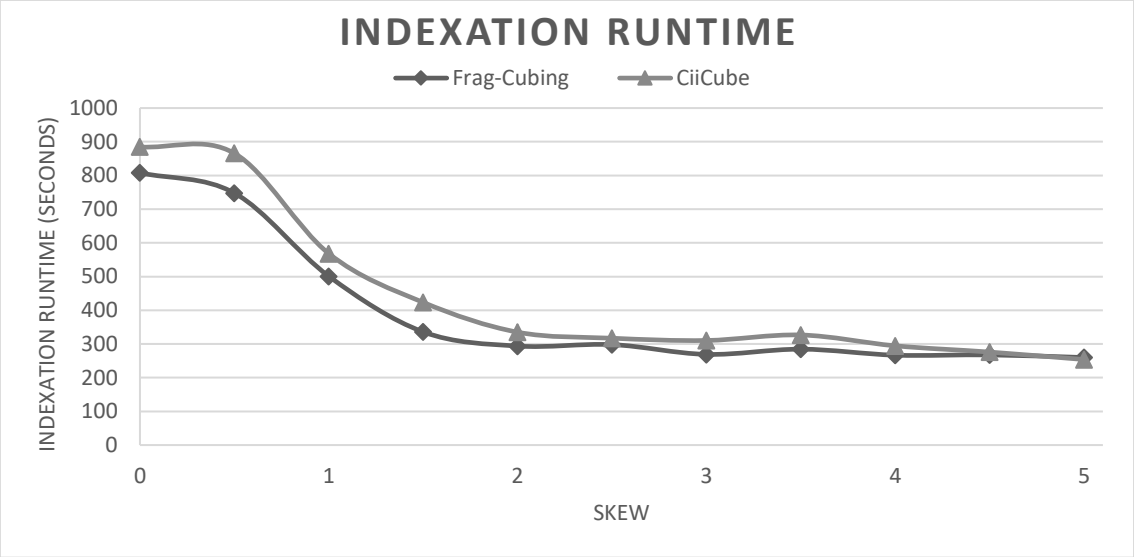


Figure 4 Indexation Runtime, in seconds, of a data set with T = 130M, D = 30, C = 2500 and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5

Regarding Structure Size, tests shown that changing the cardinality does not seem to affect considerably the memory needed in both algorithms. Both algorithms had the linear memory consumption increase following the increase of dimensions and tuples, however, interesting results, as expected, are obtained when changing the skew.

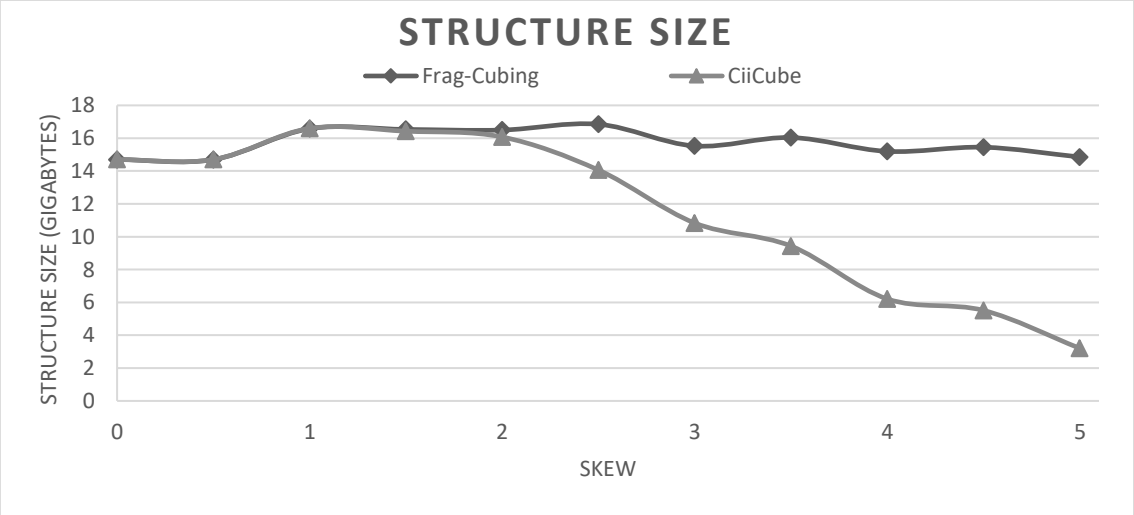


Figure 5 Structure Size, in Gigabytes, of a data set with T = 130M, D = 30, C = 2500 and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5

In *Figure 5* it is possible to see the evolution of both algorithms' structure size varying the skew. When the skew is above 1.5 the compression can be done more effectively and the CiiCube algorithm starts using considerably less memory than Frag-Cubing. Notably, when the S = 5, the CiiCube algorithm utilizes around 25.6% of the memory that Frag-Cubing needs. The higher the data set skew, the higher the chances of the same values

being repeated in following tuples. The more the same attribute value gets repeated in following tuples, the better CiiCube's compression algorithm works, hence these results.

An unrelated test was made using a data set with data set with T = 500 million, D = 30, C = 2500 and S = 5. CiiCube index such data set with an indexation runtime of 16.4 minutes and a structure size of 11.99 Gigabytes. The Frag-Cubing algorithm was unable to index the same data set using the 30 gigabytes of main memory available in our system.

4.3. Point Queries

Some tests comparing both Frag-Cubing and CiiCube answering point queries were done. To these tests we created used a data set with T = 130 million, D = 30 and C = 2500. The point queries made had 30 equal operators and used the most common values so that the point queries done would have the biggest runtime possible.

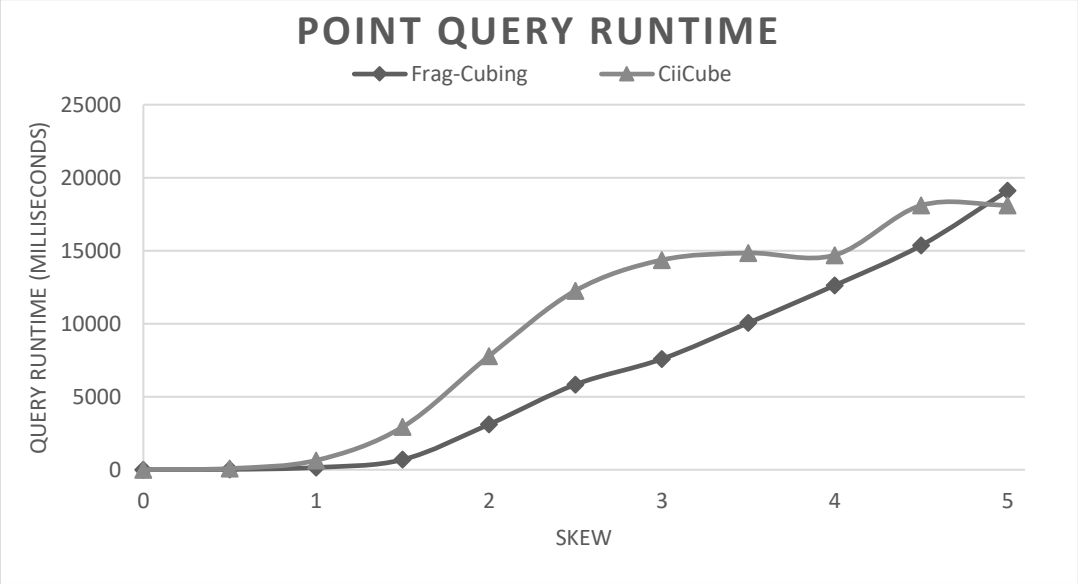


Figure 6 Point query runtime varying the Skew with T = 130M, D = 30, C = 2500, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5, and 30 Equal Operators

As it can be seen in the *Figure 5*, in general, CiiCube is slower than Frag-Cubing to answer point queries, notably, when S = 2, Frag-Cubing was more than twice as fast as CiiCube. This result comes from the fact that CiiCube's intersection algorithm needs to choose the TIDs to be compared before doing the comparison, as stated in section 3.3. The Frag-Cubing intersection algorithm does not need to do that step.

Another remarkable result is when S = 5, where CiiCube is slightly faster than Frag-Cubing. This is the result of high levels of compression that allows the algorithm to process multiple TIDs in a single comparison.

Another interesting observation is the variety in CiiCube's query runtime. When S > 2, the runtimes of the Frag-Cubing algorithm grow fairly linear, while the runtime of the CiiCube algorithm, while also having a growth tendency, grows quite inconsistently, for example, with S = 3, 3.5 and 4, the runtimes of the point queries made are approximately the same, while, with Frag-Cubing, the runtime of the point query when S = 4 was almost double the runtime when S = 3. This behaviour from CiiCube is the result of the success the algorithm had when compressing the data set.

4.4. Subcube queries

Query tests were used to assess CiiCube's performance when compared with algorithm. Two metrics were made to do such comparisons:

- Query Runtime: this metric is used to analyse the amount of time needed for both algorithms to answer a query being made;
- Query Memory Usage: this metric is used to analyse the amount of memory necessary to answer a query being made. The value here presented includes the value of the structure size plus the memory used to store the query and any secondary structure used to that effect.

CiiCube and Frag-Cubing have an approximately linear growth following the growth in number of dimensions and tuples. The cardinality does not seem to change the query runtime in both algorithms.

Just like in the indexation runtime of the data cube, the only characteristic that seems to differentiate CiiCube from Frag-Cubing is the skew of the dimensions.

In the *Figure 6* and *Figure 7* it is possible to see tests done varying the data set skewness. The data set had the following characteristics: T = 130 million, D = 30 and C = 2500. The skews tested ranged between 0 and 5, with increments of 0.5. The subcube queries done had two inquire operators and one equal operator, where its value uses one of the most recurrent attributes.

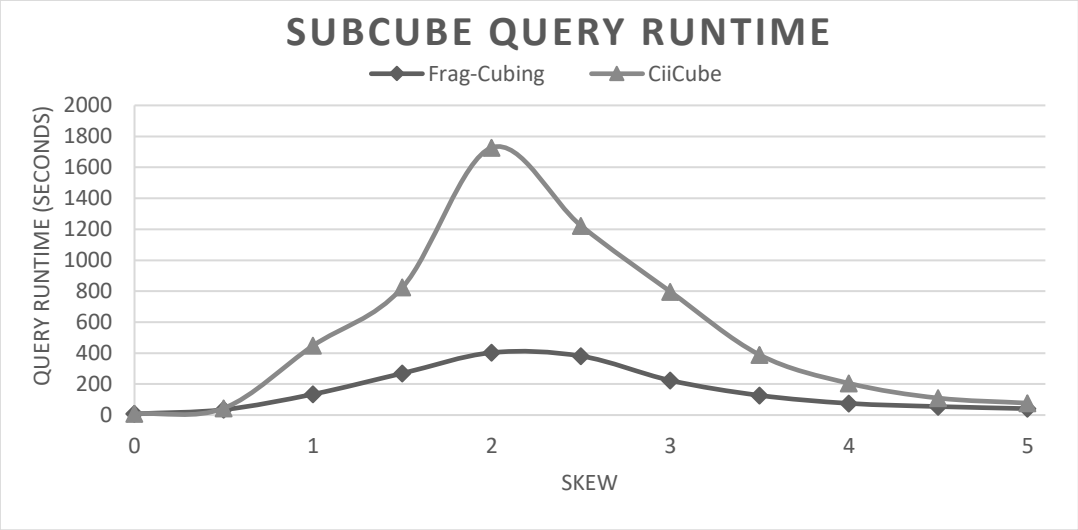


Figure 7 Subcube query runtime varying the Skew with T = 130M, D = 30, C = 1, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5, 1 Equal Operator, 2 Inquire Operators

As it is possible to see, both algorithms have a higher runtime when S=2 When S = 0 the CiiCube algorithm had almost the same runtime as Frag-Cubing, which suggests that, when compression is minimal, both algorithms have the same behaviour. The main drawback of the CiiCube algorithm is the much bigger runtimes when S is in the interval [1.5 ; 4]. This kind of difference can only be attributed to the intersection algorithm. While the CiiCube intersection algorithm needs to choose, for each *DIntArray* object being intersected, which value to use, the Frag-Cubing algorithm does not. This step seems to be the main difference in both algorithms and prime reason to this gap runtime.

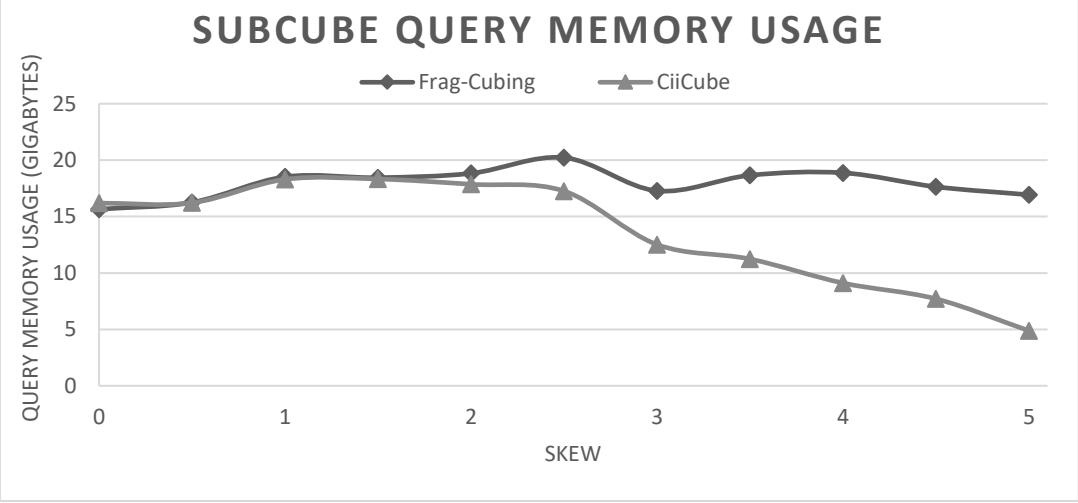


Figure 8 Subcube query memory usage varying the Skew with T = 130M, D = 30, C = 1, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5, 1 Instantiated Operator, 2 Inquire Operators

Regarding query memory usage, as it is possible to the see in *Figure 7*, increasing the skew of the data cube helped considerably to reduce the memory needed to process the

data, nonetheless, must be noted that a big part of this difference comes directly from the results of the structure size after the operation of indexation.

4.5. Tests With Real World Data

Tests with real-world data sets were also made. In this section we will exhibit the results of indexation tests and subcube queries using real-world data sets.

4.5.1. Data Sets Used

In our real-word tests we decided to use data sets with different sizes and cardinalities. Relatively to the data sets sizes, we wanted to represent small data sets, medium size data sets and big data sets. Relatively to the cardinality, we wanted to represent 3 kinds of data sets: one with low cardinality, another with high cardinally and a last one to represent a mix of both cardinalities.

We understand that concepts such as “big” or “small”, used above to categorize the data sets, are abstract, however the technique used to define those values was based of empiric observation on the data sets we had available.

The “Connect-4” [19] data set was used to represent small data sets with low cardinality. This data set contains positions in the game “connect-4”, it is composed by 67557 tuples with 42 dimensions of the which one has a cardinality of 4 and all the remaining have a cardinality of 3.

The “Forest Covertype” [20] data set was used to represent medium sized data sets with mixed cardinality. This data set has 512.012 tuples with 54 attributes with the following cardinalities: 3858, 361, 67, 1398, 801, 7118, 255, 255, 255, 7174, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, and 7.

The “Exame” [21] data set was used to represent large data sets with high cardinality. It was composed a combination of several similar COVID-19 data sets from five different hospitals in Brazil. This data set is composed by 26651926 tuples with 9 different dimensions with the cardinalities: 562943, 975188, 425, 97, 1986, 2355, 63778, 105, and 2064.

4.5.2. Data Set Forest Covertype

When indexing the “Forest Covertype” data set, the average Indexation Runtime of Frag-Cubing was 1976 milliseconds, while the CiiCube algorithm indexed the data set in 1779 milliseconds. Its Structure Size was, in Frag-Cubing, 143 Mbytes, while, using the CiiCube algorithm, was 42.7 Mbytes.

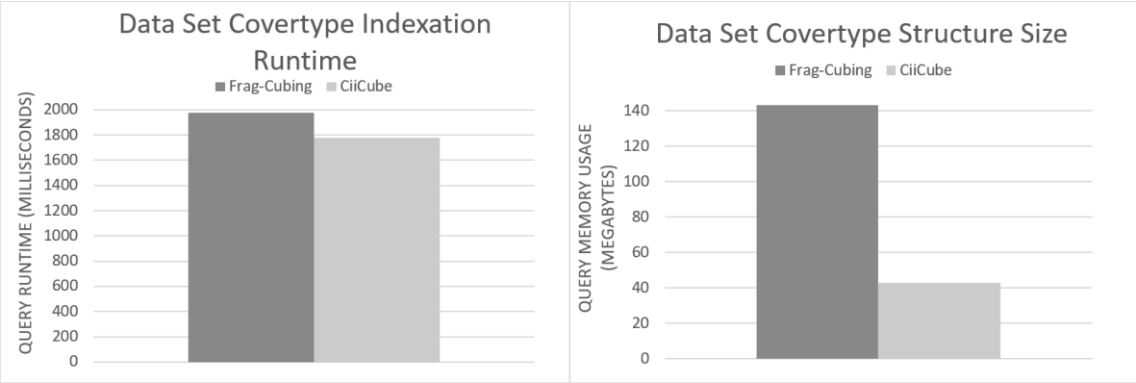


Figure 9 Forest Covtype Data Set Indexation Runtime and Indexation Structure Size

This positive result to CiiCube's indexation algorithm comes from the 40 attributes with cardinality of two and an extremely high skew, that allows to do compression quite effectively.

Three different subcube queries were done using the *Forest Covertype* data set.

The first subcube query inquired three dimensions, with cardinalities of 2, 2 and 7, respectively. The Frag-Cubing program needed 152 milliseconds and used 172.3 MBytes of memory to answer it, while the CiiCube algorithm needed 179 milliseconds and used 64.6 MBytes of memory.

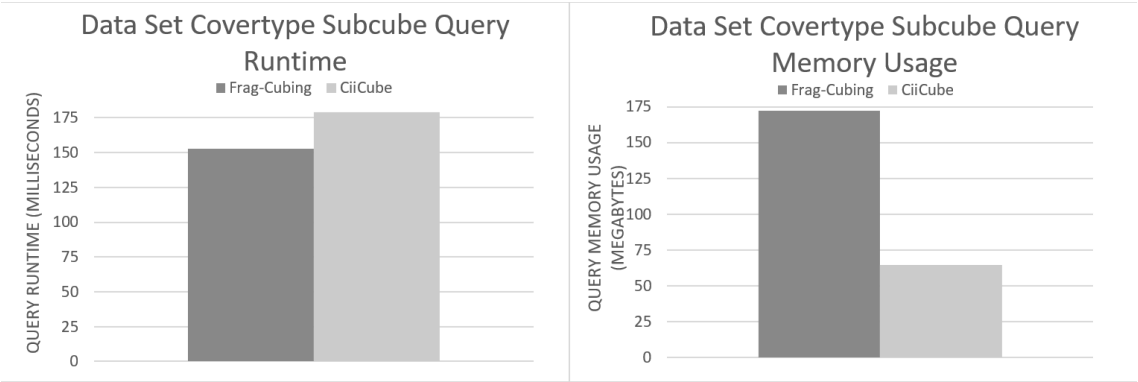


Figure 10 Forest Covertype Data Set Subcube Query Memory Usage and Query Runtime on three dimensions with C =2, 2 and 7

The second query also inquired three dimensions, with cardinalities of 3858, 1398 and 801, respectively. Frag-Cubing needed 22 seconds and used 8.53 Gigabytes of memory to answer it, while the CiiCube algorithm needed 35 seconds and used 8.21 Gigabytes of memory.

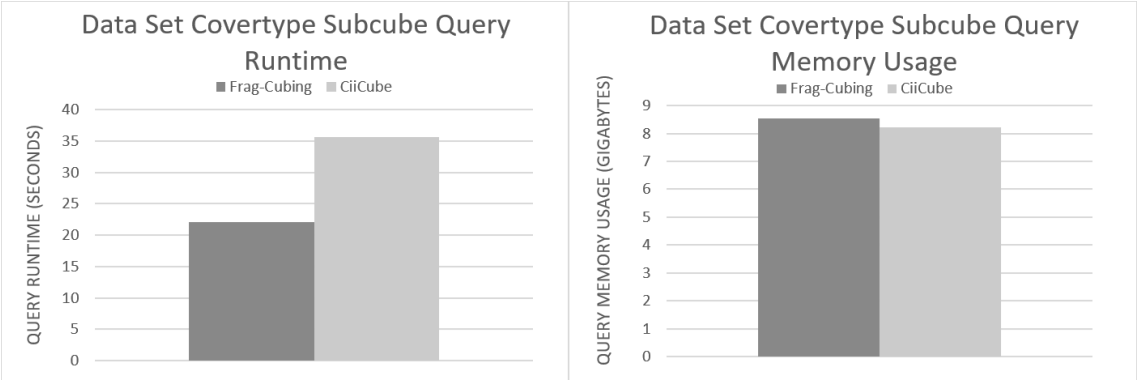


Figure 11 Forest Covertype Data Set Subcube Query Runtime and Subcube Query Memory Usage on three dimensions with C =3858, 1398 and 801

The third query inquired four dimensions, with cardinalities of 3858, 1398, 2 and 2, respectively. Frag-Cubing needed 190.6 seconds and used 12.15 Gigabytes of memory to answer it, while the CiiCube algorithm needed 156.1 seconds and used 11.99 Gigabytes of memory.

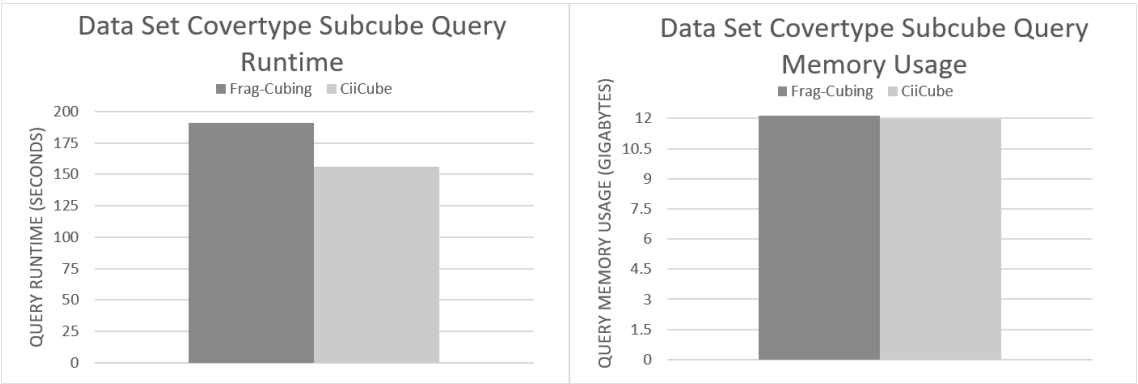


Figure 12 Forest Covertype Data Set Subcube Query Runtime and Subcube Query Memory Usage on three dimensions with C =3858, 1398, 2 and 2

The first query made was quite small, both in runtime and memory usage, indicating that Frag-Cubing has a slightly better runtime to answer this kind of subcube query. Also, CiiCube needed only almost one third of the memory used by Frag-Cubing to do that operation.

The second subcube query saw CiiCube need almost the double the runtime Frag-Cubing used to answer the query. In this case, the memory used was almost the same. This quey shows a situation where CiiCube can have a poor runtime performance, most likely due to only being able to have small compressions, resembling the teste case where the skew was equal to 2, seen in section 4.4. It is also possible to see that, to both algorithms, the query memory usage was multiple times bigger than the memory used to

store the data cube. This happened due to the massive number of possibilities queried in the subcube query, reducing the relative difference in memory usage between both algorithms.

The third query joined low and high cardinality dimensions. In this query, CiiCube had a runtime around 20% smaller than Frag-Cubing, possibly caused by the big advantage it had when intersection the low cardinality and high skew inquired dimensions. The conclusions obtained from the memory consumption in this query are the same as the one obtained from the second query.

4.5.3. Data Set Exame

The indexation of the "Exame" data set into a data cube took, for Frag-Cubing, 16.4 seconds and used 1037 megabytes, while CiiCube took 17.7 seconds and used 616 megabytes.

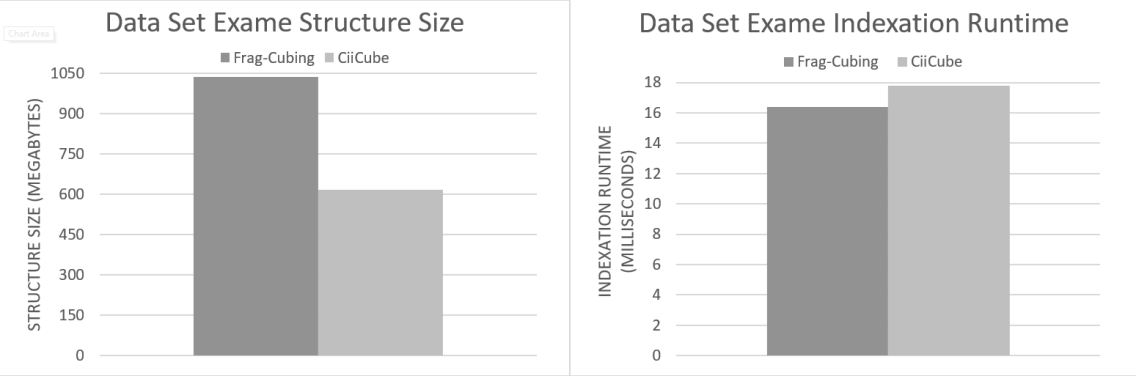


Figure 13 Exame Data Set Indexation Runtime and Indexation Structure Size

In this case, the CiiCube algorithm was able to use almost half the memory that Frag-Cubing needed to the data cube's structure. Since the data set has a great number of tuples, there are many chances to compress TID intervals, therefore reducing its size. Relatively to the indexation runtime, as expected from the tests in section 4.2, CiiCube has a slightly higher indexation runtime.

A subcube query with one equal operator, with the most common value, in the dimension with 97 cardinality and two inquire operators in the dimensions with cardinality 1986 and 2355 was done. Frag-Cubing needed 108 seconds and 2.34 gigabytes to answer it, while CiiCube needed 436.4 seconds and 1.68 gigabytes to do the same operation.

Figure 14 Exame Data Set Subcube Query Runtime and Query Memory Usage With 1 Equal Operator and 2 Inquire Operators

The difference in memory usage when answering the subcube query comes, mostly, from the initial structure size. Nonetheless, CiiCube was able to increase that difference due to the compression also used in the subcube created. CiiCube's runtime, however, is more than 4 times greater than Frag-Cubing. This result indicates that, although the compression did work, the intervals compressed had, in general, a small size, which is quite hurtful to the algorithm's runtime.

4.5.4. Data Set Connect-4

Tests done showed that, in order to index the "Connect-4" data set, Frag-Cubing needed 269.67 milliseconds and 35.23 megabytes to index and store the final data cube, while CiiCube needed 274.67 milliseconds and 26.44 megabytes to do the same operation.

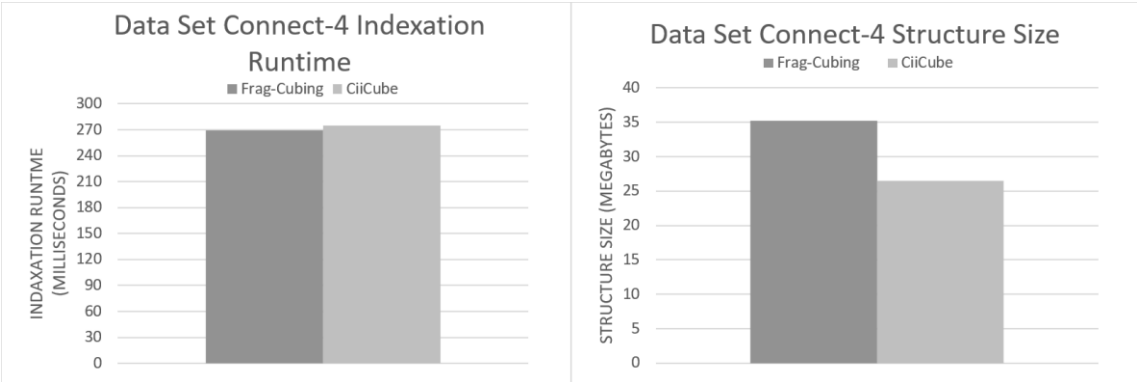


Figure 15 Connect-4 Data Set Indexation Runtime and Structure Size

Although this data set is considered small and with a low cardinality, the CiiCube algorithm managed to use around 25% less memory than Frag-Cubing. This is possible

due to the high skew existent in some of the data set's dimensions. Just like in the tests before, the indexation runtimes of both algorithms are almost the same.

The subcube query used to compare both algorithms with this data set had one equal operator, using the most common value, in the dimension with C = 4 and 10 inquire operators. Frag-Cubing used, on average, 4.75 seconds and 891.85 megabytes to answer that query, while CiiCube used 8.87 seconds and 825.67 megabytes.

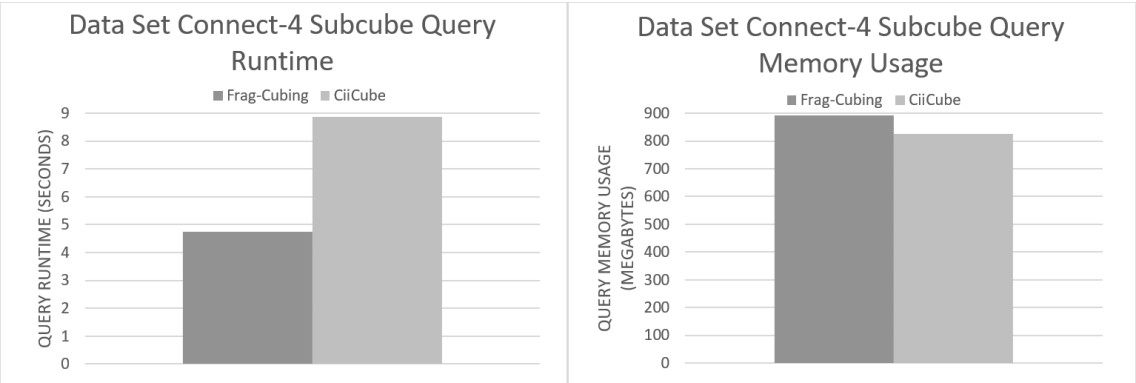


Figure 16 *Connect-4 Data Set Query Runtime and Query Memory Usage With 1 Equal Operator and 10 Inquire Operators*

The difference in memory usage is mostly caused by the difference in structure size. CiiCube's query runtime was almost the double as the Frag-Cubing query time, which has been a general rule throughout all the tests done.

5. CONCLUSIONS AND FUTURE WORK

The industry is evolving in a way that creates the need for OLAP capable algorithms that use less and less memory.

In this paper we propose a compressed inverted index data cube solution, CiiCube, which is novel way to represent inverted tuples and compared it to the well-known Frag-Cubing algorithm.

Our work was able to successfully reduce the amount of memory necessary to store and query data cubes, allowing single systems to process bigger data cubes than before. However, using this compressed structure has a runtime cost to query operations, where some queries take 4 times longer to process, when compared with Frag-Cubing.

As expected, this kind of structure is not made to completely replace the normal structure used in Frag-Cubing, since CiiCube is highly dependent on the tuple's attribute values distribution along a data set, nonetheless, it showed potential in some of the cases

tested and can be a clear choice in cases where reducing the memory consumption is prioritized over lower runtimes.

As has been many times noted during the results analysis, in subcube queries, most of the memory usage difference comes from the structure size, being made very little compression in most subcubes. It also has been noted that the CiiCube's intersection algorithm is slower than the Frag-Cubing intersection algorithm. Having that in mind, as future work, it would be interesting to, when doing subcube queries, use a non-compressed subcube, allowing for faster operations over the subcube.

References

1. Gupta, D.; Rani, R. A study of big data evolution and research challenges. In *Journal of Information Science,* 2019, 45, 322–340. doi: 10.1177/0165551518789880.
2. Garrigós-Simón, F.; Sanz-Blas, S.; Narangajavana, Y.; Buzova, D. The Nexus between Big Data and Sustainability: An Analysis of Current Trends and Developments. In *Sustainability*, 2021, *13*, 6632. doi: 10.3390/su13126632.
3. Wang, L. Design and implementation of a distributed OLAP system. In proceedings of the 2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2935-2938. doi: 10.1109/AIMSEC.2011.6010846.
4. Hellman, M. A Cryptanalytic Time-Memory Tradeoff. IEEE, 1980, 26, 401-406. doi:10.1109/tit.1980.1056220.
5. Salley, C. and E. Codd. Providing OLAP to User-Analysts: An IT Mandate, 1998.
6. Li, X.; Han J.; Gonzalez H. High-dimensional OLAP: a minimal cubing approach**.** In proceedings of the International Conference on Very Large Data Bases, 528–539. doi: 10.1016/B978-012088469-8/50048-6.
7. Beyer, K.; Ramakrishnan R. Bottom-up computation of sparse and iceberg cube. In Proceedings of the 1999 ACM SIGMOD international conference on Management of data, 359-370. doi: 10.1145/304182.304214.
8. Silva, R.R.; Hirata, C.M.; Lima, J.C. qCube: efficient integration of range query operators over a high dimension data cube. In *journal of Information and Data Management*, 2013, 4, 469–482.
9. Leng F.; Bao Y.; Yu G.; Wang D.; Liu Y. An Efficient Indexing Technique for Computing High Dimensional Data Cubes. In: Yu J.X., Kitsuregawa M., Leong H.V. (eds) Advances in Web-Age Information Management. WAIM 2006. Lecture Notes in Computer Science, vol 4016. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11775300_47
10. Ferro A.; Giugno R.; Puglisi P.L.; Pulvirenti A. BitCube: A Bottom-Up Cubing Engineering. In: Pedersen T.B., Mohania M.K., Tjoa A.M. (eds) Data Warehousing and Knowledge Discovery. DaWaK 2009. Lecture Notes in Computer Science, vol 5691. Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-03730-6_16.

11. Silva, R.R.; Hirata, C.M.; Lima, J.C. A Hybrid Memory Data Cube Approach for High Dimension Relations. . In Proceedings of the ICEIS 2015 - 17th International Conference on Enterprise Information Systems, vol 2. doi: 10.5220/0005371601390149.
12. Leng F.; Bao Y.; Yu G.; Wang D.; Liu Y. An Efficient Indexing Technique for Computing High Dimensional Data Cubes. In: Yu J.X., Kitsuregawa M., Leong H.V. (eds) Advances in Web-Age Information Management. WAIM 2006. Lecture Notes in Computer Science, vol 4016. Springer, Berlin, Heidelberg. doi: 10.1007/11775300_47.
13. Silva, R.R.; Hirata, C.M.; Lima, J.C. Big high-dimension data cube designs for hybrid memory systems. Knowl Inf Syst, 2020, 62, 4717–4746. Doi: 10.1007/s10115-020-01505-9.
14. Phan-Luong, V. A simple and efficient method for computing data cubes. In Proceedings of the 4th Int. Conf. on Communications, Computation, Networks and Technologies INNOV, 50-55. 2015.
15. Phan-Luong, V. A simple data cube representation for efficient computing and updating. In *International Journal On Advances in Intelligent Systems*, 2016.
16. Phan-Luong, V. Searching data cube for submerging and emerging cuboids. In Proceedings of the 2017 IEEE International Conference on Advanced Information Networking and Applications Science, 586–593. doi: 10.1109/AINA.2017.77.
17. Phan-Luong, V. (2019) First-Half Index Base for Querying Data Cube. In: Arai K., Kapoor S., Bhatia R. (eds) Intelligent Systems and Applications. IntelliSys 2018. Advances in Intelligent Systems and Computing, vol 868. Springer. doi: 10.1007/978-3-030-01054-6_78.
18. Phan-Luong, V. A Complete Index Base for Querying Data Cube. In: Arai K. (eds) Intelligent Systems and Applications. IntelliSys 2021. Lecture Notes in Networks and Systems, vol 295. Springer. doi:10.1007/978-3-030-82196-8_36.
19. Tromp, J. Connect-4 Data Set. URL: http://archive.ics.uci.edu/ml/datasets/Connect-4 (accessed on 15 June 2021).
20. Blackard, J.A.; Dean, D.J.; Anderson, C.W. Forest Covertype Data Set. URL: https://archive.ics.uci.edu/ml/datasets/covertype (accessed on 15 June 2021).
21. Exame Data Set. URL: https://repositoriodatasharingfapesp.uspdigital.usp.br/handle/item/2(accessed on 15 June 2021).